Smart Compilers for Reliable and Power-efficient Embedded Computing

by

Reiley Jeyapaul

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2011 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Sarma Vrudhula
Lawrence Clark
Charles Colbourn

ARIZONA STATE UNIVERSITY

May 2012

ABSTRACT

Thanks to continuous technology scaling, intelligent, fast and smaller digital systems are now available at affordable costs. As a result, digital systems have found use in a wide range of application areas that were not even imagined before, including medical (e.g., MRI, remote or post-operative monitoring devices, etc.), automotive (e.g., adaptive cruise control, anti-lock brakes, etc.), security systems (e.g., residential security gateways, surveillance devices, etc.), and in- and out-of-body sensing (e.g., capsule swallowed by patients measuring digestive system pH, heart monitors, etc.). Such computing systems, which are completely embedded within the application, are called embedded systems, as opposed to general purpose computing systems. In the design of such embedded systems, power consumption and reliability are indispensable system requirements. In battery operated portable devices, the battery is the single largest factor contributing to device cost, weight, recharging time, frequency and ultimately its usability. For example, in the Apple iPhone 4 smart-phone, the battery is 40% of the device weight, occupies 36% of its volume and allows only 7 hours (over 3G) of talk time. As embedded systems find use in a range of sensitive applications, from bio-medical applications to safety and security systems, the reliability of the computations performed becomes a crucial factor. At our current technology-node, portable embedded systems are prone to expect failures due to soft errors at the rate of once-per-year; but with aggressive technology scaling, the rate is predicted to increase exponentially to once-per-hour.

Over the years, researchers have been successful in developing techniques, implemented at different layers of the design-spectrum, to improve system power efficiency and reliability. Among the layers of design abstraction, I observe that the interface between the compiler and processor micro-architecture possesses a unique potential for efficient design optimizations. A compiler designer is able to observe and analyze the application software at a finer granularity; while the processor architect analyzes the system output (power, performance, etc.) for each executed instruction. At the compiler micro-architecture interface, if the system knowledge at the two design layers can be integrated, design optimizations at the two layers can be modified to efficiently utilize available resources and thereby achieve appreciable system-level benefits. To this effect, the thesis statement is that, “by merging system design information at the compiler and micro-architecture design layers, smart compilers can be developed, that achieve reliable and power-efficient embedded computing through: i) Pure compiler techniques, ii) Hybrid compiler micro-architecture techniques, and iii) Compiler-aware architectures”. In this dissertation demonstrates, through contributions in each of the three compiler-based techniques, the effectiveness of smart compilers in achieving power-efficiency and reliability in embedded systems.

To

Mom and Dad

ACKNOWLEDGEMENTS

"...when you have done the will of God you may receive what is promised." Heb 10:36. Praise be to the Lord God Almighty, for His loving hand has guided me over all the hurdles of life, and has made this dissertation possible.

I remember well the first day I met Prof. Aviral Shrivastava at his office, a week before the first day of classes in the Fall of 2006. After a very brief discussion on what little I had on my resume, he proceeded to present research problems and steer me into research. I didn't then, nor do I now, know what he really saw in me that led him to believe I would get so far. Prof. Shrivastava has always believed in me, and overly patient with all the havoc I created. He has taught me all I know about research, and everything that goes with it. Though I once had no intentions of pursuing a doctoral degree, I believe now that my decision to do so has been the best ever; and I owe it all to my guide Prof. Shrivastava. Thank you sir.

Prof. Jongeun Lee, joined our lab as a post doctoral researcher with an abundance of experience and expertise. I was one of the privileged few to work closely with him on a soft error project, which was to be an integral part of my thesis. His guidance, and expertise were instrumental in helping us formulate the Cache Vulnerability Equations. I take this opportunity to thank Prof. Lee for being a good friend and colleague, during his days at CML.

One of the most inspiring classes I ever had at ASU was Advanced VLSI Design by Prof. Lawrence T. Clark. His enthusiasm and work that lead to the design of tomorrow's reliable processors, has influenced a significant part of my dissertation. I am grateful to Prof. Clark for his support and guidance, throughout the duration of my tenure at ASU, both as a Masters student and also as a PhD student.

I would like to thank Prof. Charles Colbourn, for introducing me to "Design Theory" and its potential to be integrated in different parts of my research work. I again thank Prof. Colbourn for his guidance in shaping my dissertation.

I thank Prof. Sarma Vrudhula, for his support during my internship at Intel, through the Embedded Systems Consortium. As a grad student on the 4th floor, with a cubicle close to Prof. Vrudhula, I take pride in being continually motivated by him, to push the limits and strive for higher goals. Thank you sir for being a fountain of inspiration to us all.

Being a member of the Compiler Microarchitecture Lab (BY407) is a privilege in its own right.

I am lucky to be surrounded by friends, rather than colleagues, who have been a pillar of strength to me, both on and off the lab. I am but sure that our friendship will not die down after we are hooded, but will continue and possibly flourish further. I take this opportunity to thank my friends and fellow CMLers: Yooseong Kim, Ke Bai, Jing Lu, Mahdi Hamzeh, Bryce Holton, Jared Pager, Abhishek Rhisheekesan, Fei Hong, Jian Cai, Seung Chul Jung, Di Lu, and the many other alumni. From weekly meetings that go on for hours in technical discussions, to team lunches, and hour long useless discussions in the lab, I am very proud to have been with this enthusiastic bunch of researchers. Thank you guys for your support.

My brother Leontius Vinod Jeyapaul, has been my friend, philosopher and guide throughout my life. It was he who directed me to pursue Electronics engineering, and then again exposed me to graduate studies in the US. His constant support, advice, and love has been my bread and butter through these years and I am ever grateful. Though we stay miles apart, we are with each other. I owe my PhD and progress to him.

As I completed higher secondary education, my Mother (Mercy Jeyapaul) and Father(Pushpam M. F. Jeyapaul) had a *dream*. They wanted to see the prefix "Dr." before my name. Today, as I write these words on my PhD dissertation, I remember them and all the sacrifices they had to endure to make this possible. My gratitude knows no bounds, but I make one prayer – "that I may utilize what little I have gained, and make them proud". Mom and Dad, this is for you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Embedded Processors in Tomorrow's Devices

Thanks to continuous technology scaling, intelligent, fast and smaller digital systems are now available at affordable costs. As a result, digital systems have found use in a wide range of application areas that were not even imagined before, including medical (e.g., MRI, remote or post-operative monitoring devices, etc.), automotive (e.g., adaptive cruise control, anti-lock brakes, etc.), security systems (e.g., residential security gateways, surveillance devices, etc.), and in- and out-of-body sensing (e.g., capsule swallowed by patients measuring digestive system pH, heart monitors, etc.). These are applications which work on batteries intended to last long, and with a level of reliability that is considered a critical factor in its design. In addition, because of the superior power-efficiency observed in embedded technology, several high-end data servers are now being designed using embedded processors. For example, *Silicon Graphics* and *SeaMicro* have developed high-end servers using Intel Atom low power processors (*SGI Molecule* with $10,000$ cores and the *SM10000* with 512 cores). The applications of embedded processor technology in such a wide and growing array of applications shines light on the importance of *green computing*, where the computation and system are considered *reliable*.

1.2 Need for Power-Efficient Embedded Systems

Power-efficiency has been of paramount importance for embedded systems. This is because most embedded systems are mobile and therefore battery-operated, and the battery is the single most important factor that directs device cost, weight, recharging time/frequency and ultimately the usability of the system. For example, in the *Apple iPhone 4* smart-phone, the battery is 40% of device weight, occupies 36% of its volume and allows around 7 hours (over 3G) of talk time. In every generation of the *iPhone* production line, battery-life has been the most criticized feature despite of the increase in computation capabilities or applications made possible. In Appendix B, we present experimental deductions to substantiate our claim of battery life being affected by the power consumption of the CPU in a hand-held device (e.g., smartphone). A larger battery could increase the talk time (or device usage time), but would affect the device weight and thus its handling and consequentially the cost. The trade-off in such system designs typically is between the recharge frequency and system weight. Mobile devices such as these have now become a requirement rather than a luxury, to empower the fast pacing economy. Efficient embedded systems thus have to be developed, that meet the high processing requirements of such mobile devices, at low power to meet their battery specifications. *Intel's Core 2 Mobile processor* [47] and soon-to-be-released mobile processors like *Texas Instruments' OMAP 5* dual-core processor, *Nvidia's Kal-El* quad-core processor, and *Qualcomm's snapdragon* 2.5GHz quad-core processor are examples,

from the industry, that show the trend of increasing processor speeds together with the number of cores in embedded processors. It is thus evident that there is an immediate need for high-performance but power-efficient embedded processors.

1.3 Need for Reliable Embedded Computing

Another system metric that is becoming increasingly important in embedded systems, as an effect of technology scaling, is reliability. As embedded systems find use in safety-critical applications, including medical, missile navigation controls, adaptive driver assistance, and financial systems, reliability becomes a crucial system-design parameter. Soft errors are becoming the most important threat to system reliability. Soft errors are transient faults that can happen due to several reasons, including electrical noise, external interferences, cross-talk. However, majority of soft errors in the system happen due to high-energy or charge-carrying particle strikes on the digital device that toggle its logical value. Charge carrying particles like alpha particles and high energy neutrons (100KeV - 1GeV from the cosmic background) have been known to cause soft errors for a long time. With technology scaling, even low energy neutron particles (10meV - 1eV) cause soft errors in sub 45nm SRAM memory cells [96]. This effect is multiplied with the fact that there are many more low-energy particles, than those at higher energies [46]. In Appendix A, we extrapolate the SER of current mobile and general purpose processors, for future technology dimensions, and justify the urgent need for soft error research in embedded systems. At our current technology, high-end mobile systems e.g., smart-phones, are prone to experience a soft error at the rate of about *once-per-year*. With drastic technology scaling (when devices are fabricated in dimensions smaller than the wavelength of light used to produce them), and even faster increase in processor computing requirements, this rate is only expected to increase exponentially [54].

1.4 Role of Compilers in Enabling Reliable and Power-efficient Embedded Systems

Power-efficiency and reliability are well recognized concerns, and techniques to improve them have been developed at several levels of the processor design abstraction, e.g., at the manufacturing and materials (e.g., use of high-K dielectric [90], etc.), transistor level (e.g., gate sizing [28]), circuit design (e.g., multiple-threshold circuits [103]), micro-architecture design and implementation (e.g., improving branch predictor [80]), software library [84], and system level (operating system level Dynamic Voltage and Frequency Scaling (DVFS)[56] and Dynamic Thermal Management (DTM)[13]).

These design layers can be coarsely divided into hardware and software levels. Techniques at hardware level are very useful since they affect the entire chip, and do not require any change in the software stack, and consequently are very popular. For example, using high-K dielectric reduces the leakage of the whole chip, and does not require any software change. The forte of hardware techniques is in

optimizing the hardware for use in a given functionality. However, optimization techniques that apply to only parts of the application functionality are challenging to implement in hardware. This is because applying such hardware schemes requires building some intelligence in hardware to detect the application pattern to decide when to selectively apply the technique; and this extra hardware could outshine the benefits of the optimization technique. For example, if we want to reduce the power consumption by power-gating some parts of the processor when the ILP is low, then the circuitry needed to determine the ILP will execute all the time, and that may annul the benefits.

Cases when the optimization technique must be applied selectively on some parts of the application are best dealt at the software level, where there is more control over the application and its specification. Given the hardware architecture of a processor, the software specification and the way it is executed on the hardware can be tuned to optimize power and reliability. For example, the power and performance of an application executing on a processor with caches depends heavily on the number of misses in the cache. By changing the way cache is accessed while preserving the functionality, the power and performance of the application can be optimized. Such an optimization can be done by the application programmer. However the programmer is already burdened with the exponentially increasing functionality of the application, and debugging needs, so that adding more burden of application optimization (for power, reliability, etc.) on the application programmer is not advisable or desirable.

The compiler sits at the hardware software interface, between the processor micro-architecture and software layers. The input of a compiler is the application specification in a programming language, e.g., C, C++, python etc. and the output is machine instructions that are directly executed on the processor hardware. Ultimately, the compiler decides the way an application uses the processor micro-architecture, and therefore can very significantly affect the power, performance and reliability of application execution on the processor. The power of the compiler in achieving a balanced and favorable trade-off, between the various system metrics that are affected by the application, can be explained with the help of an example in Figure 1.1. Two different loop orders (IJ and JI) of a simple program have been shown executing over a fully-associative cache (LRU replacement policy) of size equal to 4 cache lines. Analysis of the program's behavior over the cache, show that though they implement the same functionality, there exists a distinct difference in their power and reliability metrics.

- In the loop order IJ (Figure 1.1(a)), for the first four iterations, the array `A`'s data is accessed *row-first* and only one cache-line is being accessed. This use of the temporal-locality of the data-cache reduces the runtime of the program and also reduces power consumption by reducing the number of memory accesses. On the other hand, we observe here that the data elements of array `A` accessed

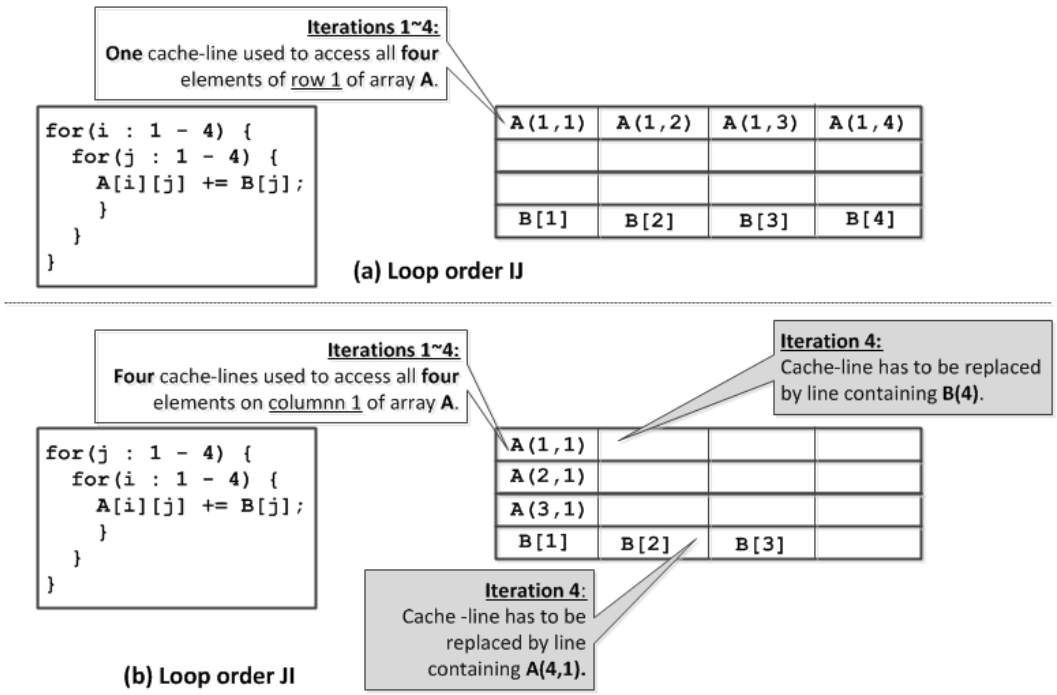


Figure 1.1: Example program showing compiler analysis on trade-offs: (a) Loop order IJ accesses data *row-first* on the array, and requires only one cache-line during the first 4 iterations. This has reduced number of memory accesses and power, but by causing data to remain in the cache (exposing them to soft errors) has increased vulnerability. (b) Loop order JI accesses data *column-first* on the array, and requires 4 cache-lines during the first 4 iterations and causes a cache-miss (access to `B(4)`). This has increased number of memory accesses, cache misses and power, but reduced vulnerability by reducing time that data is in the cache.

during the first four iterations remain in the cache till the end of the program, even if they are no longer required for program execution. The increased exposure (to soft errors) of the data-cache elements, increases the probability of soft errors and thereby reduces the reliability of the program. *This configuration thus has high power-efficiency but low reliability.*

- In the loop order JI (Figure 1.1(b)), for the first four iterations, array `A`'s data is accessed *column-first*, where the memory arrangement is such that the elements of each column occupy separate cache-lines. For the first three iterations, three cache-lines are loaded into the cache (accessing columns $1, 2$, and 3). On the fourth iteration, since the cache is of limited size, when the fourth array element on column 4 is to be accessed, the cache-line containing elements of array `B` has to be evicted. In addition, during the same iteration an access to `B(4)` requires that the first cache-line that contained row 1 of array `A`, be evicted based on the LRU replacement policy. During the first 4 iterations of the program, this configuration incurs a total of 4 memory accesses and one cache-miss which leads to an overall increase in the number of memory accesses and energy consumption. On the other hand in this loop order, the accessed cache-lines are quickly replaced by

a new cache-line thereby reducing the time cache data is exposed to soft errors. *This configuration thus has high reliability but low power-efficiency.*

In terms of optimization, the compiler has the advantage that the whole application code is available to it, and there is time (since compilation is usually off-line) for deep analysis (as seen in the example above). This marks the larger set of resources (application software and system hardware knowledge) available to the compiler, which is one of the primary considerations to choose this layer of abstraction. One important manifestation of this is that while hardware techniques can only have information of the recent past, a compiler can get an idea of the future, i.e., what kind of computation is coming next, and prepare the micro-architecture for the same. For example, if the ILP in the computation coming next is low, it can issue instructions to power-gate some of the circuits that are not going to be used. Finally, looking bottom up from hardware, the compiler is the last layer at which the effects of hardware changes can be absorbed, so as not to affect the software stack. Consequently, it is also a place where new hardware modifications can be exploited automatically.

1.5 Summary

In summary, embedded processors are finding use over a wide spectrum of applications with an increasing trend in the processor requirements. Improving the power-efficiency and reliability of tomorrow's embedded systems are some of the most important challenges in their successful designs. Among the different abstraction layers in system design, the *compiler* is poised at a unique position between the application and the micro-architecture. It has, i) the knowledge of the application code (at a finer granularity), and ii) can analyze the effect of code on the processor micro-architecture. The compiler, with such a unique perspective, is key to the development of smart techniques to improve the power-efficiency and reliability of current and future embedded systems.

Chapter 2

THESIS STATEMENT

My thesis is that, "by merging system deign information at the compiler and micro-architecture design layers levels, smart compilers can be developed; that achieve reliable and power-efficient embedded computing through: i) Pure compiler techniques, ii) Hybrid compiler micro-architecture techniques, and iii) Compiler-aware architectures."

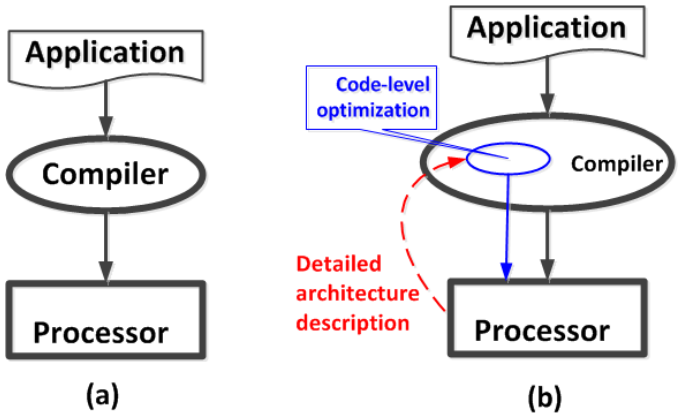


Figure 2.1: (a) **Traditional Compilers**; (b)**Pure Compiler Techniques**: Compiler optimizations using detailed micro-architecture description of the processor.

Traditionally, the role of the compiler is to translate the code of a given application (in a high-level language like "C, C++ or JAVA") to low-level machine instructions (like `load, store, mov, add`, etc.) as per the processor ISA (Instruction Set Architecture), which is represented by Figure 2.1(a). In this case, the compiler optimizations (if any) are machine-independent and are targeted to improve the overall performance of the application or reduce code-size.

Advanced code-level optimizations can be developed to affect certain specific system metrics like power and/or reliability. Using the micro-architecture level description of the underlying processor, a given application can be analyzed for its impact on the system metrics and optimized thereof. Such techniques, that aim to improve the power-efficiency and reliability of a system, using only compiler optimizations on the application code, are called **Pure Compiler Techniques** and represented by Figure 2.1(b).

Some hardware techniques use specialized circuitry, or modified architecture blocks, to reduce the power consumption or increase the reliability of the system. Details on the behavior of such hardware blocks can be used by the compiler to develop code-level optimizations, which enable efficient utilization of these modified architecture blocks. Such combined software-hardware techniques used, to improve the

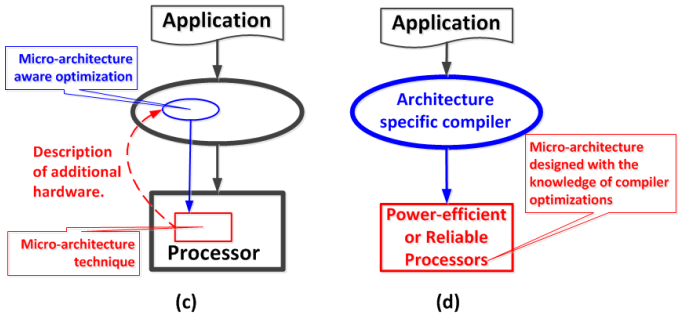


Figure 2.2: (c)**Hybrid Compiler Micro-architecture Techniques**: Compiler optimizations are targeted to efficiently utilize specific architecture blocks; (d)**Compiler Directed Architectures**: New power-efficient (or reliable) processors are developed in conjunction with the compiler designed for the specific micro-architecture.

power-efficiency or reliability achieved (by the hardware technique alone), are called **Hybrid Compiler Micro-architecture Techniques** and represented by Figure 2.2(c).

To reduce the power consumption or increase reliability in the processor, the core architecture can be greatly modified (for e.g., the use of many-core architectures for power and performance). However, the use of such architecture depends on the specific compiler capabilities that can compile (and/or optimize) a given application for that processor. The key idea is that, by reducing processor hardware and complexity power consumption is reduced, while the task of intelligently executing code on the simplified architecture is shifted to the compiler. For example, to run an application on the low-power IBM CELL processor an architecture specific compiler is required and to improve performance at low power requirements, specific optimizations have to be employed. Such processor architectures designed in conjunction with the compiler, by allowing migration of the processing complexity to the compiler, are called **Compiler Directed Architectures** and represented by Figure 2.2(d).

Chapter 3

CONTRIBUTIONS OF THIS DISSERTATION

Our[1] contributions listed here demonstrate the application of the proposed *thesis* to current and future embedded system design.

3.1 Pure Compiler Techniques

In the compiler, in addition to the processor ISA, details on the behavior of certain architecture blocks (like the cache, branch target buffer, register file, re-order buffer, etc.) are required for developing compiler optimizations to improve the power-efficiency and reliability of the system. For example, by knowing the cache size, associativity and replacement policy, the program data access patterns can be modified to reduce cache-misses (memory accesses) and thereby increase performance and reduce power consumption. Similarly, exposure of data cache contents to soft errors can be analyzed through compiler techniques and accordingly the reliability of the system can be optimized. In this dissertation,

- we develop a static analysis technique: *Cache Vulnerability Equations*, that can estimate the vulnerability (extent of soft error exposure) of a given program, and help choose the right code transformations that improve the program's reliability.

3.2 Hybrid Compiler Micro-architecture Techniques

For Reliability

During program execution, updated data elements (dirty cache lines) as they reside on the cache are exposed to soft errors and therefore have an increased probability of data corruption. A micro-architecture technique, *Cache Scrubbing* [76] proposes to periodically visit data elements in the cache and write copies of the same into the memory. This approach helps to reduce the lifetime of updated data in the cache, and thereby increases system reliability. However with an increase in the number of cache write-backs, power is consumed in the memory and therefore a trade-off exists between reliability and power.

- In the compiler, by analyzing the memory profile of the application, we identify specific instances to perform reduced number of cache write-backs and thereby reduce power consumption but still achieve the intended system reliability.

For Power Efficiency

Among the architecture blocks in a processor, the TLB (Translation Lookaside Buffer) is one of the most prominent power consuming components. The reasons for such a behavior are: i) for every cache

[1] In order give due credit to my collaborators in the work that contributed to this dissertation, I present the work in the collective.

access, the TLB is accessed for page translation, ii) the TLB resides on the cache-access critical path, and iii) the TLB is composed of dynamic (highly power consuming) circuitry. One very efficient micro-architecture technique that achieves significant power reduction (as a standalone technique) in the TLB is the *Use-Last* TLB architecture[37].

- We develop instruction scheduling and code transformations to modify the data access patterns to ensure improved power savings in the data-TLB, over that achieved by the *Use-Last* TLB architecture.
- We again develop code placement techniques implemented post-pass, to reduce power consumption of the instruction-TLB using the *Use-Last* TLB architecture.

3.3 Compiler Directed Architectures

To reduce power consumption, but still improve on the performance of a system, one popular methodology used by processor manufacturers is to reduce the size and complexity of a single core and increase the total number of such cores within the processor. Therefore, the overall throughput of the system is increased while each core runs at a slower speed and lower power rating. The key idea behind such an architecture is that by reducing the hardware complexity and power consuming architecture blocks, the system power is reduced. One such low power but high performance, many-core architecture is CGRA (Course Grained Re-configurable Architecture). The problem in using such an architecture is that a given application has to be statically scheduled in time and space to execute over the processor's array structure.

- We develop a dynamic scheduling technique that enables the CGRA to be used as an accelerator to a multi-threaded processor. Our technique allows components of multiple threads to be accelerated simultaneously by the CGRA, thus enabling multi-threading in the CGRA processor.

Chapter 4

PURE COMPILER TECHNIQUES

"Through optimizations in the compiler, researchers have developed techniques to combat power efficiency [42, 98] and reliability [57, 60] issues in embedded microprocessors. These techniques are implemented in the software design layer, and are exposed to the application and only an abstracted perspective of the processor architecture. The reason for such an abstraction is to increase the portability of the designed software across a wide range of processor micro-architectures. In doing so, we observe that the lack of micro-architecture level information reduces the efficiency of the developed technique and thereby achieved benefits. In this chapter, we highlight such trade-offs between portability and overall efficiency, and propose micro-architecture specific pure-compiler optimizations for increased system reliability."

In this work, we perform a series of design space experiments to demonstrate the need for compiler directed techniques armed with micro-architecture knowledge, to improve system reliability. Through these experiments and analysis, we deduce that in the development of such micr-architecture specific pure-compiler optimizations, *estimation* becomes the key requirement. To this effect, we develop an efficient and accurate static estimation technique to measure the reliability of the system (in specific the data cache), for any given micro-architecture configuration. This enables the compiler to exploit the performance-vulnerability trade-offs in embedded applications at compile-time; and introduce specialized code transformations to efficiently improve program reliability.

4.1 Introduction

Inside a processor, memory elements are most susceptible to soft errors, not only because they are typically the largest structures by area and transistor count, but also because there is no logical and temporal masking of soft errors in memories, and they operate on lower voltage swings [15, 31, 64, 92]. In addition, owing to the operating conditions and technology dependent design of caches (at subnano dimensions), the critical charge (Q_{crit}) required to flip the data stored in a bit is also reduced [77]. In fact, according to [73], more than 50% of soft errors happen in memories. Lower levels of memories (farther from the processor) can be relatively easily be protected using ECC (Error Correcting Code) based techniques, but protecting memories closer to the core (i.e., L1 caches) results in high overheads. Previous research [62, 85] has shown that implementing SEC-DED (Singe-Error Correction and Double-Error Detection) can increase the L1 cache access latency by up to 95%, power consumption by up to 22%, and chip-area increase by up to 18%. Even if the performance overhead could be hidden, the power and area overheads cannot. Moreover, due to the impact of process variations at sub-nano

device dimensions, SEC-DED in caches is increasingly being used to cover up hard errors in the cache, leaving only parity checking available for runtime error detection in most cache blocks. The other option of implementing double-bit error correction has extremely high overheads [1, 78]. Another popular approach is to use write-through L1 caches. Write-through L1 caches ensure at least two copies of the latest data, therefore they drastically reduce the vulnerability of data in caches, but this greatly increases the memory traffic between the processor and the lower levels of memory. Consequently, they are not a desirable and scalable solution for multi-core and multi-processor systems [44].

A soft error manifests in the cache when a random extraneous charge disrupts the data stored in a bit, thereby causing data corruption in the memory. *A datum in the memory is vulnerable to SDC, if the corrupted data value is the only updated copy, and is likely to propagate through the system causing failure (system crash or erroneous output)*. Data cache *vulnerability* is thus defined as the time for which such vulnerable data bytes remain exposed to soft errors in the cache; which in turn translates into the probability of soft error induced failures in the system. At different stages of system design, various decisions and parameters affect the data cache vulnerability realized, like:

1. **Software Design** – Given a functionality, the choice of the right algorithm, its implementation and appropriate compiler optimizations, are some of the key factors associated with software design.

   (a) We explore the impact of **"algorithm choice and implementation"** on *data cache vulnerability*. In a broader sense, the runtime of an algorithm is proportional to the time that data remains in the cache. However, our experimental analysis shows that, owing to the variations in implementation (read write pattern), the vulnerability of the program does not exactly follow the same trend as that of runtime; thus demonstrating a more complex relationship that requires accurate estimation to assist in decision making.

   (b) An implemented program is generally compiled to machine code by the compiler. Through the use of **"compiler optimizations"**, various code transformation tools may be used to optimize the code for performance. Code transformations change the read/write access pattern of the program variables, and therefore effect the *data cache vulnerability*; with or without an impact on performance. However choosing the right code transformation at compile-time is an open problem, which requires extensive analysis of both program and cache architecture, and also accurate estimation.

2. **Hardware Design** – Through modifications to the cache microarchitecture, parameters like **"cache size, set associativity, and block size"**, can be changed; which directly impact the *data cache vul-*

nerability of the executed program. However, no single cache configuration can be said to have the same effect on different applications or the varying application parameters. A mechanism, to integrate the software and hardware characteristics, is required to accurately analyze and also estimate the vulnerability of the system designed.

3. **System I/O** – The size of the data used during the execution of an application is an end-user specific parameter, and is generally not considered during system design. In this paper, for the first time (to the best of our knowledge), we perform design space exploration over varying data-set sizes, across different cache configurations. The interesting relationships that are formed between system input-sizes and *data cache vulnerability*, thus lead us to the conclusion that, with the knowledge of the average range of data-sizes that an application is to be executed over, accurate vulnerability estimation techniques can assist in the right choice of cache architecture parameters and also in software design decisions.

In this chapter, we perform exploratory experiments on benchmark applications, to study the impact of the various system design parameters on data cache vulnerability. Our analysis of the results indicate that these design parameters have a strong inter-dependence affecting vulnerability of the system. We perform a range of consolidated experiments where appropriate design choices are made for a given embedded application, and the compiler directed code transformations that can be implemented are changed. In this, we study the maximum possible vulnerability variation achievable and their corresponding performance trade-offs, for each benchmark. We observe that, to be able to find design/execution points which are good both in terms of runtime and vulnerability (thereby exploiting the available trade-offs) is difficult. For this, we need a method to estimate the vulnerability of data in caches accurately, including the inter-dependency between the application's data access pattern, and the underlying cache microarchitecture. Only cycle-accurate simulation based techniques are known to estimate cache vulnerability accurately. While they can certainly be used (e.g. in our exploratory experiments in) to explore some code transformations, and optimize for vulnerability and runtime, they are limited by the overall time taken to analyze the entire code-transformation design space. Thus there is a need for efficient techniques to estimate cache vulnerability of programs. To this end, in this paper, we develop analytical techniques to estimate data cache vulnerability in caches. In spite of analytic techniques being efficient, they provide insights which can be used to develop simpler techniques to approximate the data cache vulnerability.

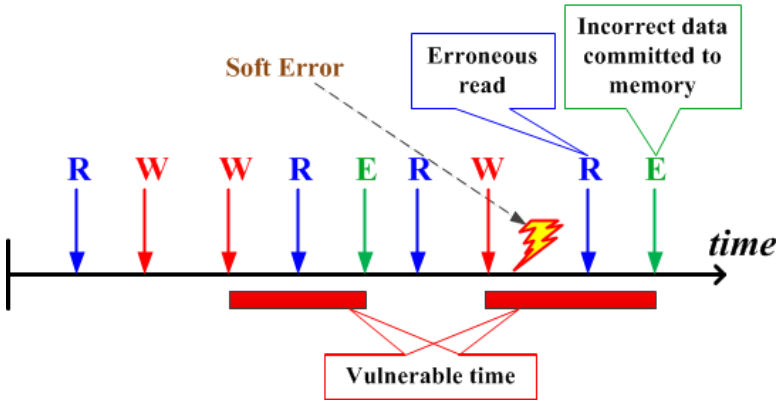


Figure 4.1: Data Cache Vulnerability (Example): Cache operations on a *datum* are denoted by labeled arrows (R-Read,W-Write,E-Eviction) over its timeline in the cache. Below this timeline, the vulnerable time-regions are marked by solid bars. In the presence of an SEU (denoted by a lightning-bolt), the following read (R) and/or eviction (E) events operate over incorrect data, and carry forward the same into the underlying memory.

4.2 Cache Behavior: Data Accesses and Vulnerability

Based on analysis by Mukherjee et al. [76], we observe that the probability of double-bit errors is negligible (typically 3 orders of magnitude lesser), in comparison to single-bit errors. In such a case, we may assume that simple hardware techniques (most likely implemented in current processor implementations) like cache-line bit-interleaving can reduce the onset of multi-bit errors in caches. As in the case of popular processor architectures like Intel Xscale® [48], Intel IA-32® [49], AMD Athlon® [3], etc., we assume that the L1 cache is enabled with parity-based error detection hardware [38]. In this example, and the remainder of the analysis and experiments in this paper, we adhere to the above assumptions.

In the presence of parity-based 1-bit error detection in the L1 cache, data corrupted during its time in the cache can be detected when read/accessed. If the data was only read, it can be corrected by reloading from the memory. However, if the data in the cache was written (dirty), it becomes the only correct copy of the value updated by the program. In Figure 4.1, the vulnerable time-periods on the timeline are highlighted. If an SEU corrupts the data during this *vulnerable time-period* (shown by a "bolt" on the timeline, in Figure 4.1), the program accesses and/or updates the memory with the incorrect data (which has been corrupted by a SEU) from the cache. This action may lead to system failure, or at the least cause incorrect data output from the application.

The total time that each *datum* remains vulnerable in the cache, is its *vulnerability*. The sum of the vulnerability values of each data element in the cache (accessed by the program), is the data cache *vulnerability* of the program. The *vulnerability* metric that measures the reliability of a system, may

be visualized (in an orthogonal perspective) as: *the probability of a soft error corrupted data causing system failure (or data corruption); which is directly proportional to the vulnerability (in time) of the data in the cache.*

In our analysis, we estimate the total vulnerability (in byte-cycles) and runtime (cycles) of the program. We then calculate the *Average Instantaneous Vulnerability*(AIV) of the program (measured in bytes), which is given by Equation (4.1). In other words, AIV is equal to the total number of data bytes that are vulnerable, in the cache, per cycle; which is averaged over the total runtime of the program.

$$Average\ Instantaneous\ Vulnerability(AIV) = \frac{Total\ Vulnerability(byte-cycles)}{Application\ Runtime(Cycles)} \tag{4.1}$$

4.3 Cache Vulnerability: Need For a Smart Compiler

The reliability of a program is related to its data cache vulnerability, which in turn is dependent on the time that the program's data spends in the cache. Various factors, at different stages of design abstraction like: i)software (algorithm choice, compiler optimization), ii)hardware (cache architecture parameters), and iii)system input data-set sizes, affect the time and/or quantity of data that is accessed within the cache. In this section, we attempt to study, with the help of exploratory experiments, the effects of the various "controllable or modifiable" factors that affect the data cache vulnerability of a program. In addition, we also study the means to "control or modify" the said parameters and achieve vulnerability reduction. Our analysis of the results show that significant vulnerability reduction can be achieved by varying some combination of the influencing parameters; however, no single and generalized cache/program configuration can be confirmed to guarantee similar vulnerability results for all applications. In order to reliable systems through system design choices, an accurate estimation methodology is required which incorporates the inter-dependencies of the "controllable or modifiable" of the system.

Experimental Setup

For our explorative experiments, we model a system with a RISC processor, an on-chip L1 cache (size=32KB) and off-chip SDRAM memory. The SimpleScalar [14] `sim-outorder` cycle-accurate simulator is configured to model the Intel XScale [48] processor architecture, with a 4-way set associative L1 cache. Unless otherwise stated, we use these cache parameters to be that of the baseline cache architecture. The simulator is instrumented with code to accurately evaluate vulnerability of data used in the program (in byte-cycles). Based on analysis by Mukherjee et al. [76], we observe that the probability of double-bit errors is negligible (typically 3 orders of magnitude lesser), in comparison to single-bit errors. In our experiments, we assume the occurrence of 1-bit soft errors per cache-block (based on discussion in Section 4.2). In each set of experiments, one parameter is varied while maintaining the others constant, to analyze their independent effects on the data cache vulnerability.

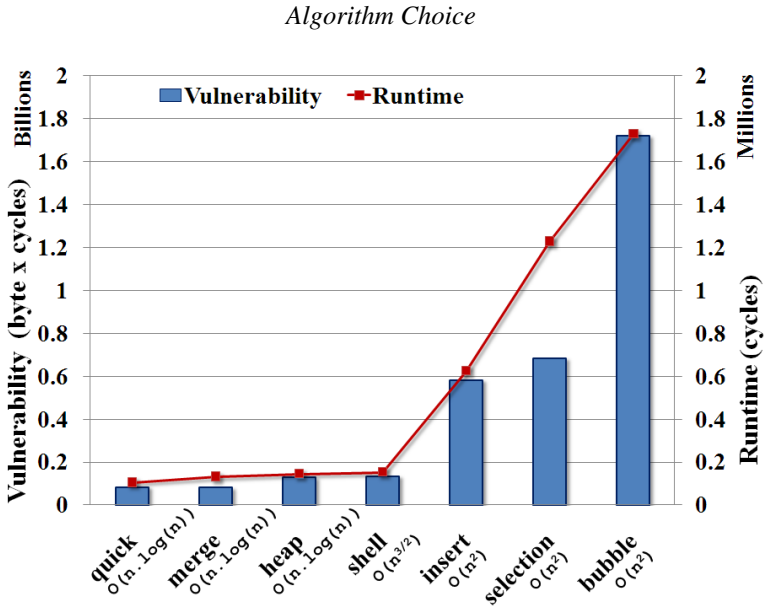


Figure 4.2: Vulnerability - Runtime trade-off analysis across `sort` algorithms. Total data cache vulnerability follows the trend of program runtime, for the same data-set, across algorithms.

In software design, one of the first design decisions would be the choice of an appropriate algorithm for the application. This choice affects the mechanism of application-data accesses and thereby impacts the data cache vulnerability of the program; and plot the results. We explore this impact at a more finer granularity through experiments over seven `sort` algorithms. In this, all the seven programs operate over the same random data-set, and the application is executed stand-alone on the baseline processor configuration. Figure 4.2 plots the total vulnerability and runtime of the algorithms; while the runtime complexities are labeled on each data point. We see here that the total vulnerability of the program follows a trend similar to that of the program runtime. On the other hand, Figure 4.3 plots the instantaneous cache vulnerability of the cache, averaged over the program runtime. We see here that for algorithms arranged in increasing runtime, the instantaneous vulnerability does not follow the same trend.

For a program that executes longer, the data utilized by the program is loaded and accessed within the cache longer; and is therefore more vulnerable. This reasoning governs the cache vulnerability plots in Figure 4.2. At any given cycle, the number of data-bytes that remain vulnerable in the cache is governed by the program's data access pattern that utilizes the data in the cache at each cycle. Therefore, as shown in Figure 4.3, the individual algorithms and their implementations, govern their respective instantaneous vulnerabilities. In other words, for two or more algorithms with comparable runtimes,

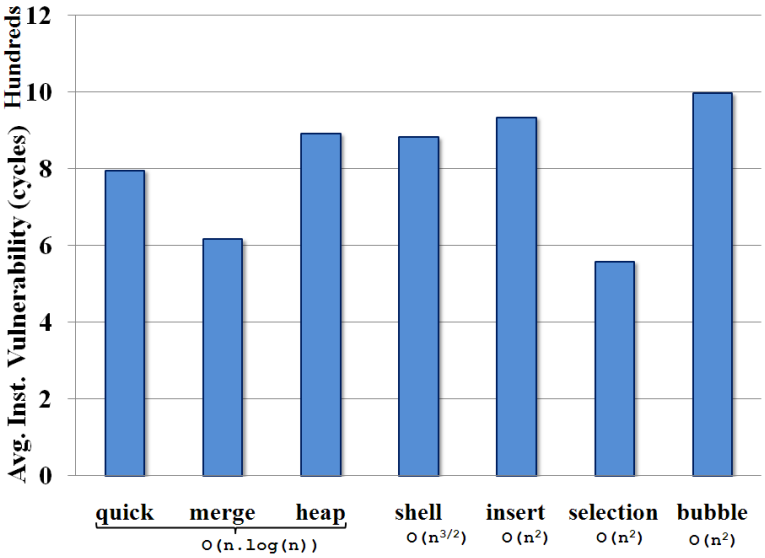


Figure 4.3: Average Instantaneous vulnerability (AIV) plot across `sort` algorithms. The number of vulnerable data-bytes per cycle (or AIV), is a characteristic of the algorithm design; and does not follow the trend of program runtime.

a measure of the average number of bytes that remain vulnerable at each cycle, gives a mechanism to distinguish between them based on reliability. The measured AIV of a program helps to analyze the intricate relationship between the data access patterns of the program, and the data cache vulnerability initiated by the same. Therefore, data cache vulnerability (or reliability) of a given system, cannot be affected by an algorithm choice based on runtime alone; but only by a intricate and accurate mechanism that measures vulnerability, while also considering the relationship between the applications data access pattern and cache usage.

Compiler Optimizations

Having chosen a program implementation to be used in the system, the same can be optimized for performance, code size, etc., at the compiler using optimization switches. In this, static analysis on the code evaluates the performance of the program for a given ISA, and code transformations are implemented. The code transformations like: loop interchange, loop fusion, and data layout transformations like array interleaving, and array placement, can change the read/write pattern of program variables in the cache. When calibrated correctly, these transformations can have a significant effect on the vulnerability of data in the cache; with or without performance penalty. In this section, we consider one of these code transformations – *loop interchange*, and study its effect on data cache vulnerability for a program. Loop interchange is when the loop orders of a nested loop are modified to vary the order of data accesses by the program; without any change to the program functionality. For example, in the *matmul* – matrix mul-

tiplication program, the three loop dimensions can be interchanged in six different ways, and each have an impact on the data access pattern. In this, the application involves three 2D arrays of 32×32 words each (total data-set size = 12KB), and 4-way set associative cache of size 4KB size (baseline cache configuration). L1 data cache vulnerability is measured using cycle-accurate simulation. Figure 4.4 shows the vulnerability and runtime results for all six loop orders.

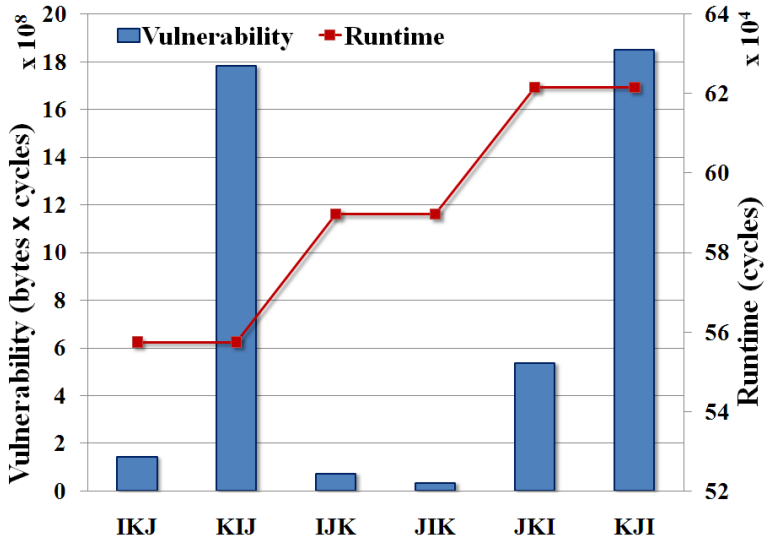


Figure 4.4: Runtime and data cache vulnerability for loop interchange on `matmul`. A significant variation in vulnerability ($53\times$) is observed between loop orders JIK and KJI, for only 5% performance variation. Loop order JIK demonstrates lowest total vulnerability among all loop orders, and is $2\times$ less than that of IJK, with $< 1\%$ performance penalty.

The **first observation** from the graph is that there is a much greater variation in vulnerability ($53\times$, from JKI to KIJ) than in runtime (11%, from KIJ to KJI). This shows that there is an interesting trade-off between vulnerability and runtime – in the sense that vulnerability can be significantly reduced at low runtime overhead.

The **second observation** we make from this graph is that the JIK loop order has relatively low runtime, and also low vulnerability. In fact as compared to the least runtime loop order IKJ, we increase runtime by around 5%, while reducing the data-cache vulnerability by more than $4\times$. This motivates for the need of finding such design/execution points, which simultaneously optimize runtime and vulnerability.

Our **final observation** from the graph is that the trend of runtime and vulnerability is not dependent, and cannot be derived from one another. This is a little counter-intuitive since to a first order of approximation, increase in runtime should imply an increase in the vulnerability, since the data spends

more time in the cache. However, vulnerability depends on many other factors including program's data access pattern, cache parameters, and data placement. This alternate perspective can be further analyzed by observing the plot of AIV in Figure 4.5, for the same six `matmul` loop orders. The trend of AIV does not follow that of runtime, nor does it adhere to that of total vulnerability as observed in Figure 4.4.

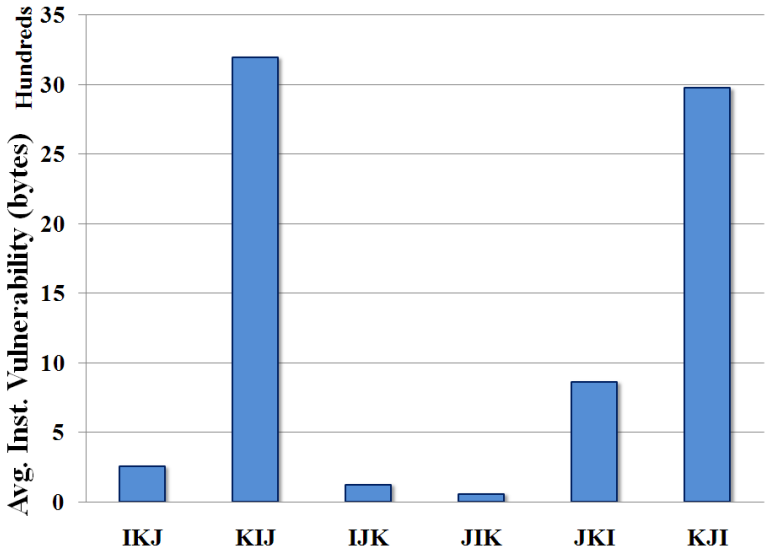


Figure 4.5: Average Instantaneous vulnerability (AIV) plot for loop interchange on `matmul`. The AIV plot for the different loop orders show the same trend as that of total vulnerability, with loop order JIK being the most economic in vulnerability and performance.

Data Size

Given an application, by varying the size of the input data-set operated on, the trend for variation in total data cache vulnerability is but obvious. However, the AIV realized by such variations in data-set size follows a non-intuitive trend. Here we plot the AIV of the `matmul` and seven `sort` algorithms while varying the data-set sizes in each.

In the case of the `matmul` benchmark (Figure 4.6), operated over a constant cache of size 32KB, we observe that,

- as the size of the operated data-set increases from $N = 10$ (1KB) to $N = 60$ (42KB), we observe a rising trend in the AIV. As long as there is enough place in the cache to retain all the data required by the program, the number of vulnerable data-bytes that are allowed to remain in the cache increases with data size.
- for data-set sizes increases beyond $N = 60$ (42KB), cache replacements come into action and therefore reduce the number of vulnerable data-bytes that are allowed to remain in the cache.

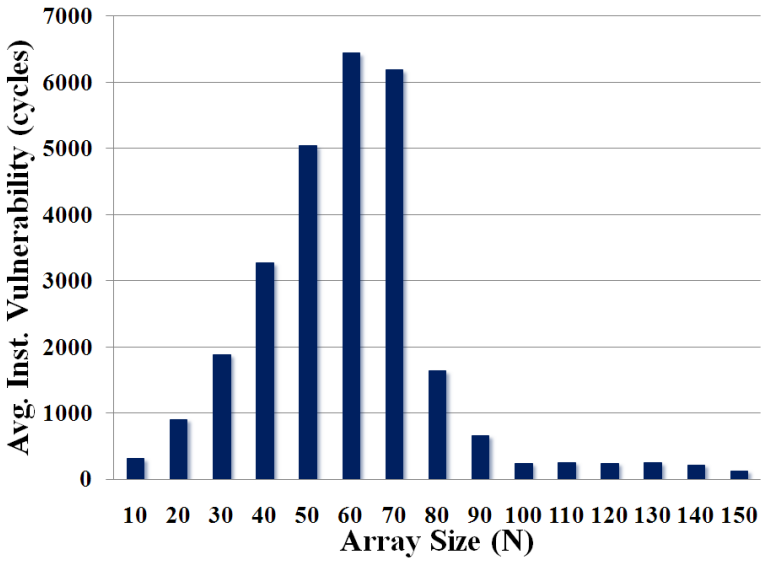


Figure 4.6: Experiments on `matmul`, by varying the input data-set size, with constant cache size of 32KB. Increase in the value of N translates to a data-set size increase of $3 \times N \times N$.

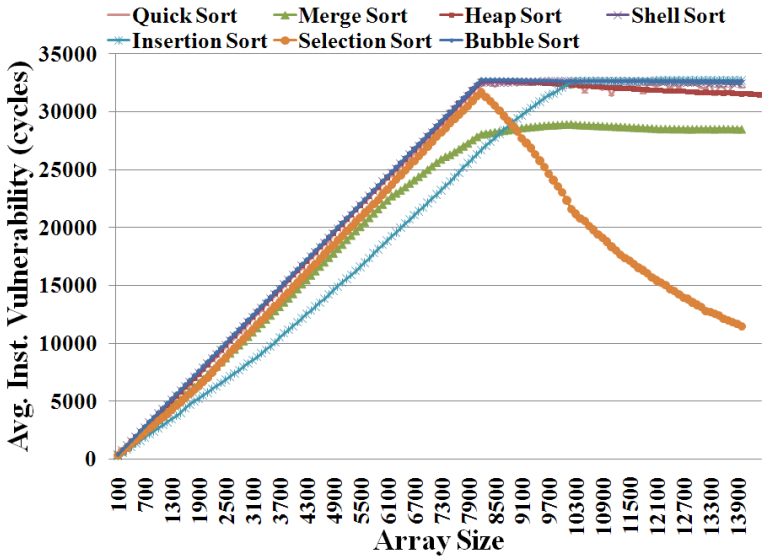
This reduces with increase in data-set size as the number of cache-replacements also increases accordingly.

- for data-set sizes beyond $N = 100$ (120KB), cache replacements occur almost every cycle and remove vulnerable data from the cache. Therefore, for data-set sizes beyond this point we observe a saturated and low AIV for the application. It should be noted here that, this configuration does suffer from very high cache-replacements thus affecting the overall performance of the system.

In the case of the `sort` algorithms (Figure 4.7), operated over a constant cache of size 32KB :

- with increase in data-set sizes from $N = 100$ (0.4KB) to $N = 8000$ (32KB), we observe an increasing trend in the AIV; similar to that of `matmul`.
- in the range of data-set size increases, from $N = 8000$ (32KB) to $N = 9000$ (40KB), we observe that different benchmarks reach their respective saturation points of AIV. This difference in their saturation bounds is completely dependent on their respective data access patterns and on the amount of array data repeatedly accessed (read, written, etc.).
- for data-set size increases beyond the saturation points, all but one algorithm follows a relatively steady AIV trend. The data access pattern of the `selection-sort` algorithm is such that, it writes into a sorted array position only once, and performs read operations only on monotonically increasing indices of the unsorted array. This behavior poses an interesting data access pattern and

Figure 4.7: Experiments on `sort` algorithms, by varying the input data-set size, with constant cache size of 32KB. The AIV for each sorting algorithm is plotted, when the size of the array to be sorted is increased.



irrespective of its increased runtime, it demonstrates comparatively lower vulnerability for larger data-sets.

From the above experiments it is evident that, for a given cache architecture, and for varying data-set sizes, the vulnerability realized by the system cannot be predicted. In addition, given an application and cache architecture, a definitive range of data-set sizes cannot be defined to guarantee a particular vulnerability trend (e.g., the anomalous behavior of `selection-sort` disrupts any trend graph drawn over the `sort` algorithms). Since the data access behavior of the application has a strong dependency on the data cache vulnerability, more accurate and perceptive (of data-size variations) means to measure the vulnerability is required; which can thereby affect system reliability.

Cache Parameters

In this section, we analyze the variation in vulnerability with respect to the cache architecture parameters like: i) cache size, ii) set associativity, and iii) cache-block size. In each, we perform experiments varying one of the cache parameters while maintaining the others constant, so as to study their independent effects.

Cache Size

The behavior of the cache in response to increasing cache sizes is very similar to that of reducing data-set sizes, for a constant cache size. Given an application operating over a constant data-set size, by increasing the cache-size of the processor, more cache space is made available to retain data longer; which

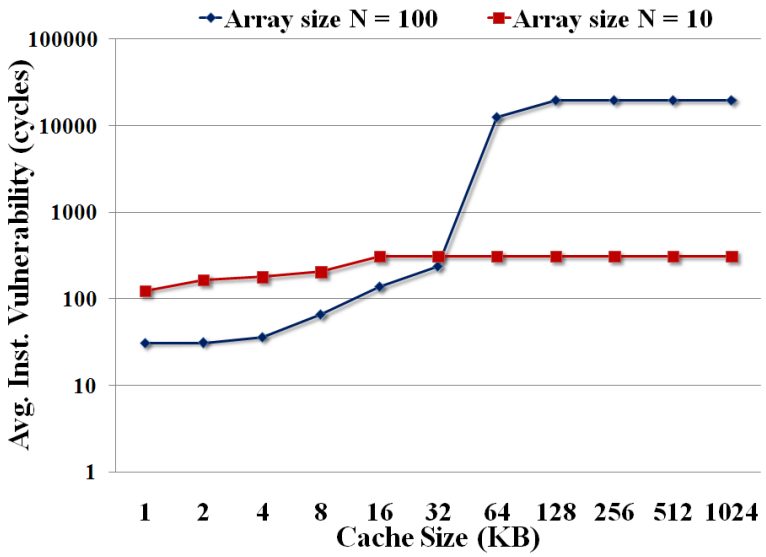


Figure 4.8: Experiments varying cache size on the `matmul` benchmark for two different array sizes N=10 (1KB) and N=100 (120KB).

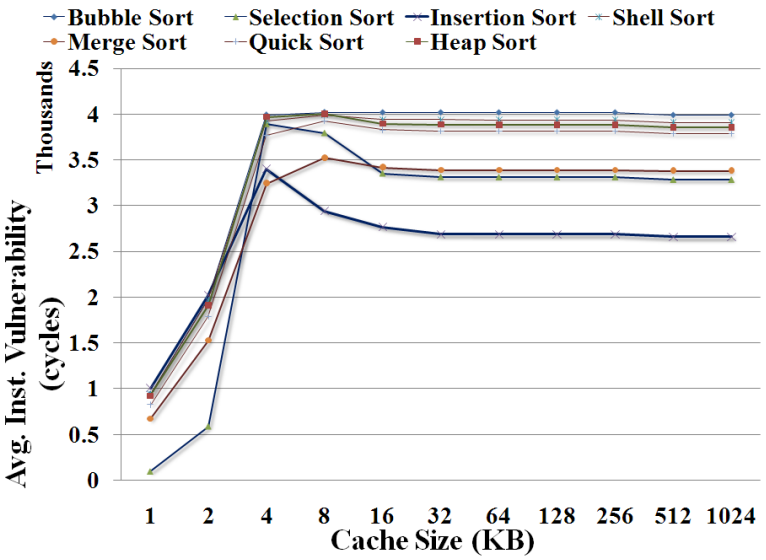


Figure 4.9: Experiments on `sort` benchmarks by varying the cache size over two different data sizes. AIV plot of `sort`; N=1000 (4KB)

thereby contributes to increased data cache vulnerability. Figure 4.8 plots the instantaneous vulnerability behavior of the `matmul` application for increasing cache sizes. We observe here that similar to that of the data size experiments, when the cache is large enough to retain all the required data by the program, the AIV saturates and follows the same trend. Similarly, Figure 4.3 and Figure 4.3 plot the AIV, for varying cache sizes, on the seven `sorting` algorithms. We again observe here a similar trend of peak AIV beyond the saturation design point (when all the data required by the program can be retained in

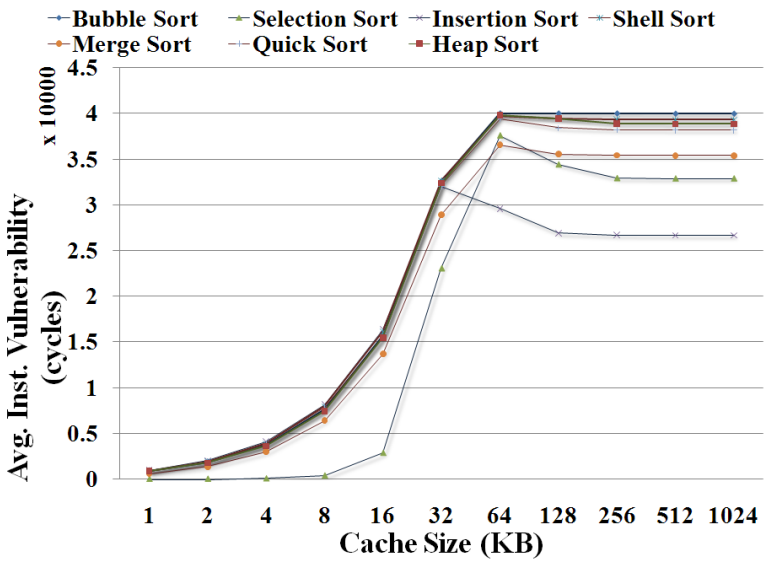


Figure 4.10: Experiments on `sort` benchmarks by varying the cache size over two different data sizes. AIV plot of `sort` N=10000 (40KB)

the cache). As in the case of data-size experiments, we see that the `selection-sort` algorithm has a slightly varied behavior, with a dip in saturated AIV, when cache size is >> data-set size. This behavior and the AIV saturations can be attributed to the dependency of the program's data access pattern, program parameters, and cache size.

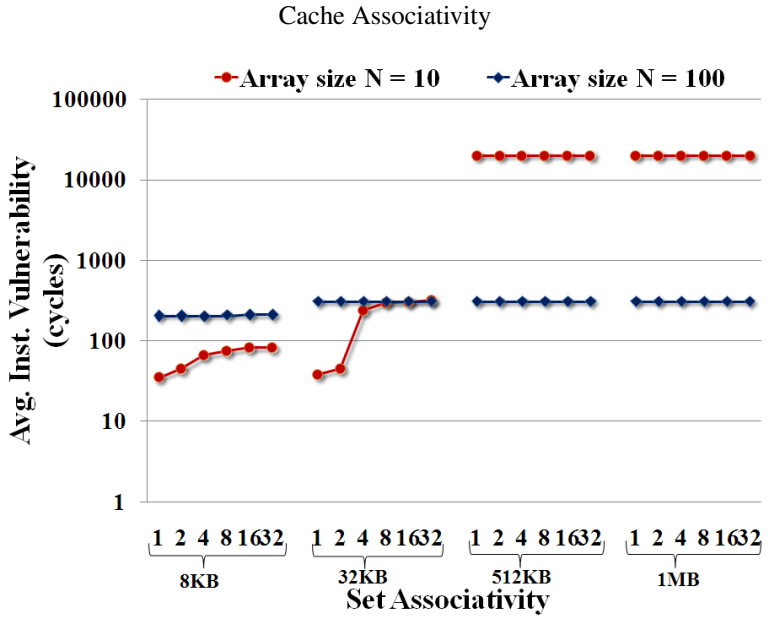


Figure 4.11: Experiments increasing the cache associativity on the `matmul` for two different array sizes.

For a given cache size and program data-set size, the instantaneous vulnerability of the program

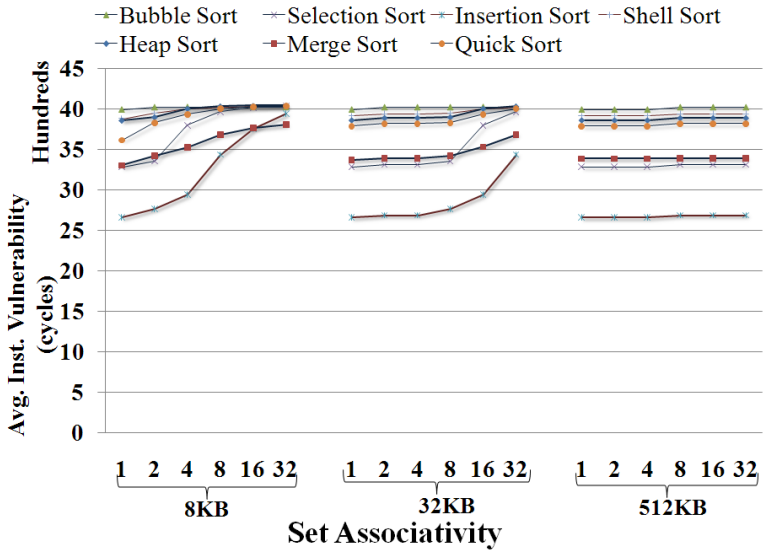


Figure 4.12: AIV plot of `sort` benchmarks varying cache associativity over two different data size configurations each. [`sort` N=1000 (4KB)]

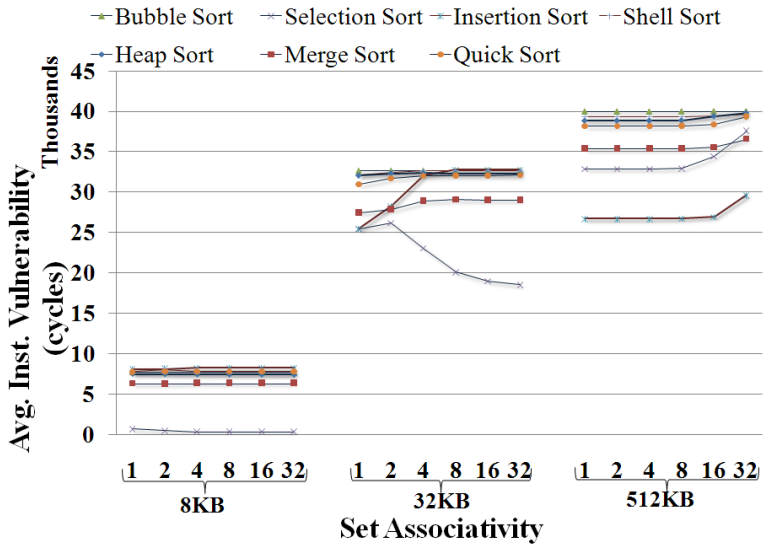


Figure 4.13: AIV plot of `sort` benchmarks varying cache associativity over two different data size configurations each. [`sort` N=10000 (40KB)]

increases with increase in set associativity. Figure 4.11, Figure 4.3, and Figure 4.3 plot the results of our cache set associativity experiments over `matmul` and `sort` benchmarks respectively. With an increase in the set associativity the number of available cache-blocks available to retain the accessed data, is increased. Though this improves on the locality of the cache, the number of vulnerable data blocks retained in the cache increases; thereby adding to the instantaneous vulnerability realized. However, this behavior cannot be generalized as the data access patterns of the application govern the intricate

use of such available cache-blocks. The vulnerability decrease of `selection-sort` with increase in set associativity (in Figure 4.3, and Figure 4.3) for cache size 32*KB* is an example of application data access pattern affecting cache vulnerability in conjunction with the change in set associativity.

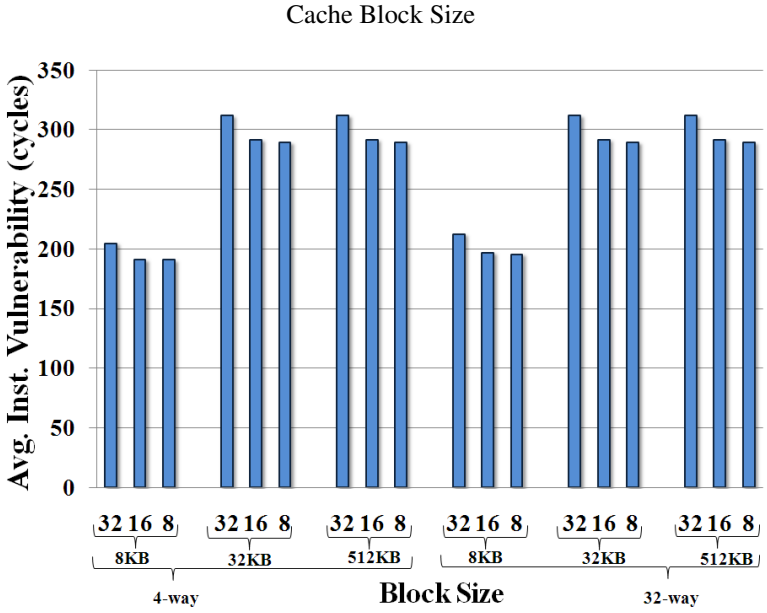


Figure 4.14: Experiments varying cache-block size on the `matmul` benchmark over two cache sizes, and three different cache associativity configurations.

For a given cache size and set associativity, if the cache-block size is increased, the number of data bytes spatially co-located to each data element is increased. With this, if one word of a cache-block is written (updated), the entire cache-block is deemed dirty and therefore all the neighboring data in the cache-block is also vulnerable. In addition, when successive data access patterns span beyond cache-block boundaries, the number of cache-blocks brought into the cache increases, thereby adding to the number of vulnerable data-bytes in the cache. Figure 4.14 plots the results of varying cache-block sizes for a given cache size, set associativity and data-set parameters. Though a trend holds good for various cache sizes and cache associativity parameters, there exist cases when the vulnerability of both 16*B* and 8*B* cache-blocks are the same. No clear and distinctive cache parameter can be derived here, as the overall impact is governed by the underlying application implementation and data access pattern.

System-level Experiments

Having analyzed each of the influential factors independently, for their effect on data cache vulnerability, in this section, we demonstrate through simulation, that interesting trade-off exists between vulnerability and runtime of applications. In this, a set of benchmark applications (important loop kernels from the SPEC 2000 and multimedia benchmark suites) are taken, and executed over a fixed cache microarchi-

tecture configuration. The benchmarks are compiled with gcc(version 2.95.3) using the '-O' option to ensure that the compiler does reschedule the loops. We consider here a L1 data cache of size 32KB, direct mapped, with 32B cache block size; which is based on a conservative selection of cache parameters, that achieve low data cache vulnerability, based on the exploratory experiments performed above. While the coupled direct relationship between runtime and vulnerability is true in general, there is a very significant impact of the data access pattern on the cache behavior; significantly changing the amount of vulnerable data is present in the cache, and therefore this direct coupling may not be realized.

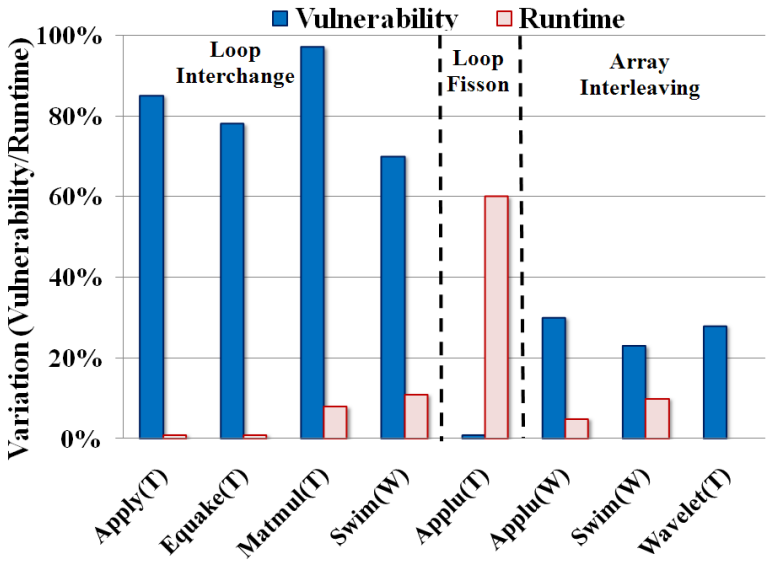


Figure 4.15: Runtime vs. Vulnerability: Opportunities to greatly reduce vulnerability at little performance cost exist.

Figure 4.15 plots the variation in the runtime and vulnerability of the $L1$ data cache for three popular loop and data transformations, *loop interchange*, *loop fusion*, and *array interleaving*. For each transformation, we find the setting that results in the minimum vulnerability and the setting that results in the maximum vulnerability. The vulnerability variation is then computed as difference in the vulnerabilities of these configuration divided by the vulnerability in the maximum vulnerability setting. The runtime variation is also computed using the minimum and maximum vulnerability settings. For example, for loop interchange on matrix multiplication, Figure 4.4 shows, that the maximum vulnerability loop order is KJI, and the minimum vulnerability loop order is JIK. The runtime and vulnerability variations are computed for these configurations. For loop fission there are only two setting, either the loops are fused, or they are separate (fission). Similarly there are two settings for the array interleaving case, either all the arrays are separate, or all the arrays in the loop are interleaved.

Next to the application names the letter, **T** or **W**, in parentheses indicates the direction of vari-

ation; vulnerability and runtime move in the opposite directions (trade-off or T) or in the same direction (win-win or W). In about half the applications (particularly for loop interchange), we observe trade-off relationship between vulnerability and runtime, typically with much less variation in runtime (46% vulnerability variation vs. 16% runtime variation, on average). This means that for some applications we can greatly reduce vulnerability while affecting performance very little, reconfirming our motivation for cost-effective soft error approaches by compilers. In a win-win situation, on the other hand, we can get automatic vulnerability reduction by choosing performance-optimal loop transformations.

Clearly the percentage variation in runtime and vulnerability is sensitive on the relative size of the application data and cache size and other cache and memory parameters. For example, if the cache is extremely small, and all accesses miss, then will be little impact of loop orders on either cache misses, or the vulnerability. Similarly, if the cache is quite large, and there are only capacity misses, then again there will be no variation in the runtime and vulnerability of the loops. However, in general, we expect the variation in vulnerability to be much more magnified than the variation in the runtime, due to the multiplicative effect of misses in vulnerability computation. To exploit vulnerability-runtime trade-offs, techniques to estimate vulnerability are required, and efficient techniques will be needed if we want the compiler to make these trade-offs automatically.

Observations and Deductions

We observe the following, based on our design space explorations over the various factors affecting data cache vulnerability:

1. An algorithm choice can be confirmed to have a better performance than another, based on runtime analysis; but one algorithm cannot be definitely concluded to have better reliability than another.

2. Code transformations on an algorithm implementation has an extensive scope for performance efficient vulnerability reduction.

3. Increase in the size of the data-set operated by an application, affects the data cache vulnerability; but does not always follow an intuitive trend. In some cases, for a known data-set range, the cache parameters and algorithm choice for least vulnerability-performance trade-off varies.

4. Increase in the size of the cache increases the available cache-blocks to retain vulnerable data and thereby increase vulnerability; but the application's data access pattern may under-use the available cache and reduce vulnerability.

5. Increasing the cache associativity expands the cache availability, improves locality and thereby allows for increased number of vulnerable cache-blocks to be retained; but the exact memory locations accessed by the program govern the use of cache-blocks within a set, and therefore can vary the vulnerability realized by the program.

6. By reducing the cache-block size, the number of data elements deemed vulnerable, when one of the elements is updated, decreases and thereby reduces vulnerability; but a program which efficiently utilizes the spatial-locality of data may have varying vulnerability estimates.

It should be noted here that any modification to the algorithm choice results in software design cost, and a change in the cache parameters results in hardware design cost. Compiler based code transformations on the other hand do not pose a design cost neither in software, nor in hardware. However, to be able to implement such a technique an efficient method, to identify design points which are good both in terms of runtime and vulnerability, has to be devised. To be able to find such design points, we need a scheme to estimate the vulnerability of data in caches. Only cycle-accurate simulation based techniques are known to estimate cache vulnerability. While they can certainly be used (e.g. in our motivating example) to explore some code transformations, and optimize for vulnerability and runtime, however there are limitations. Cycle-accurate simulation is very slow (only a few Kilo instructions-per-second [14]), and the design space for some compiler transformations can be very large. For example, using cycle-accurate simulation to explore design space for array placement for even a 32×32 matrix multiplication will take months on a 2 GHz dual-core processor system. Thus there is a need for efficient techniques to estimate cache vulnerability of programs; that accommodate the various interdependent factors influencing data cache vulnerability. To exploit vulnerability-runtime trade-offs, an accurate static analysis techniques to estimate vulnerability are required, if we want the compiler to make these trade-off decisions automatically.

4.4 Analytical Cache Vulnerability Analysis

Program Model

As is common with many static loop analysis techniques (e.g., [21, 33]), we consider a single loop nest, whose loop bounds and array index expressions are defined by affine functions of the enclosing loop indices. We also assume that all the load/store references inside a nest correspond to only the array references. Scalars can be analyzed as single element arrays. We use reuse vectors to find the last access, which further requires that references generate memory addresses in a uniform manner. In this paper we consider only perfectly nested loops and assume that the loop body has no conditional statement other than the loop itself. We also assume that memory accesses are made only through array references and

arrays do not overlap (no alias). In practice, however, the constraints in our program model are not too restrictive. Ghosh et al. [33] showed by an empirical study, that most of the runtime-wise important loop nests in standard benchmark suits, like SpecFP are amenable to these analysis constraints.

Architecture Model

The basic architecture modeled here is a uni-processor model with a single-level, data cache hierarchy. It will be possible to extend the analysis to multi-level cache hierarchies, but, lower level of caches are relatively easily and routinely protected through ECC-based hardware techniques. We assume a direct-mapped cache with write-allocate policy, implying that if the processor writes to a cache block, and the cache block is not present in the cache, it is brought into the cache before writing on it.

Terminology

A **Reference** is a static memory read or write operation in the program whereas an *access* refers to a dynamic instance of a reference [33]. In the example illustrated in Figure 4.16 (a), $R_a = a[j]$, and $R_b = b[i]$ are references. For $N = 4$, each of them is invoked 16 times, and each invocation is an access.

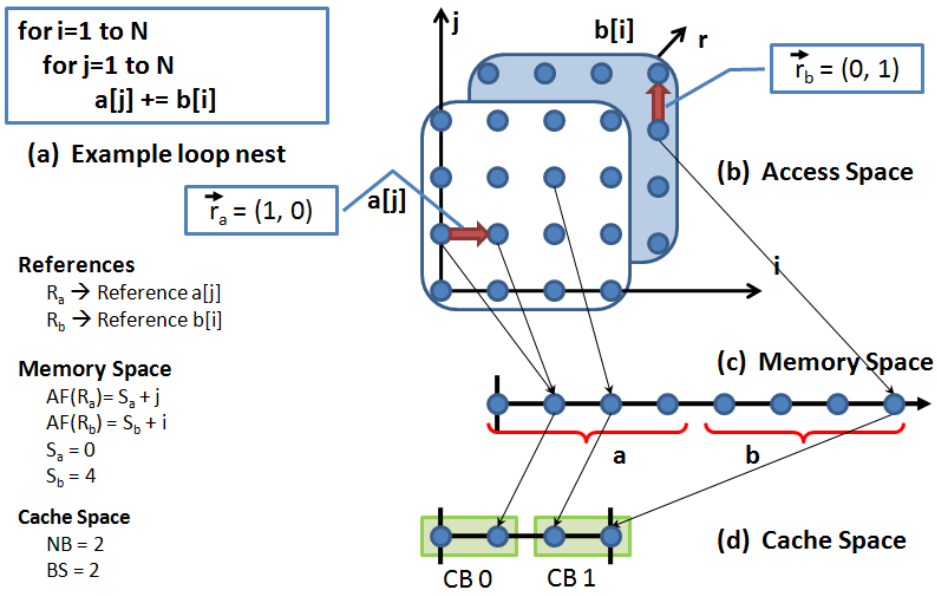


Figure 4.16: Access Space and Access Relations: **(a)** $R_a = a[j]$ is a reference, while each instance of it when the program executes is an access. **(b)** 16 points in the $i \times j$ space denotes the iteration space, and 2 sets of these points, one for reference R_a and one for R_b constitute the access space. **(c)** Accesses $(0,1,R_a)$ and $(1,1,R_a)$ have a *reuse relation* since they access the same memory address. **(d)** Accesses $(2,2,R_a)$, and $(3,3,R_b)$ access the same cache block, therefore they have a *conflict relation*, while accesses $(1,1,R_a)$ and $(2,2,R_a)$ are *unrelated* since they access different cache blocks.

The iterations of an n-level nested loop can be represented by an n-dimensional convex polytope $\mathscr{I} \subset \mathscr{Z}^n$ bounded by the loop bounds, called **Iteration Space** (outermost loop index being the first

element in the vector). Each point in an iteration space represents an *iteration* of the loop nest. We augment each iteration with reference ID to represent *access* (IDs are given in the order in which they will be accessed in the loop of interest). We then define the **Access Space** as $\mathscr{A} = \{(\vec{j}, R) \mid \vec{j} \in \mathscr{I}, R \in \mathscr{R}\}$, where $\mathscr{R}$ is the set of IDs for all references. In the example (Figure 4.16(a)), the access space of reference R_a is shown by the 4×4 grid of points in the light square, while that of R_b is shown by the points in the dark square in Figure 4.16(b).

Like iterations, accesses are ordered. An access $(\vec{j}, R)$ precedes another access $(\vec{k}, S)$ if $(\vec{j}, R)$ is lexicographically less than $(\vec{k}, S)$, or $(\vec{j}, R) \prec (\vec{k}, S)$. [1] We use access and iteration interchangeably when considering only one reference.

The mapping from an iteration to memory address for a reference is called it's *access function*, $AF_R : \mathscr{I} \rightarrow \mathscr{Z}$, and the set of all possible memory addresses accessed is the **Memory Space** of the program. Figure 4.16(c) shows the memory space of the program, where both arrays have 4 elements. Array a starts from the origin, and array b starts immediately after it ends. The access function of reference R_a is $AF_{R_a}(i, j) = S_a + j$, where $S_a = 0$ is the starting address of the array a in the memory. To model caches, we note that data is organized as blocks in caches. The *cache block* function CB gives the cache block number for a memory address. Thus $CB : \mathscr{Z} \rightarrow \mathscr{Z}$. The set of block numbers is called **Cache Space**. For a direct mapped cache, $CB(n) = (\frac{n}{CS})\%BS$, where CS is the cache size, and BS is the block size of the cache. In the example in Figure 4.16(d), the cache has 2 blocks, each of size 2 elements.

Every pair of memory accesses have either a *reuse relation*, a *conflict relation*, or are *unrelated*. Two accesses have a **reuse relation** if they access the same memory address. In Figure 4.16, the access $(0, 1, R_a)$ and $(1, 1, R_a)$ access the same address in memory (corresponding to the location of $a[1]$, therefore they have a *reuse relation*. Given a memory access, $\vec{a} = (\vec{j}, R)$, any access $(\vec{k}, S)$ that has reuse relation with it is called a *reuse access* of $\vec{a}$, and $\vec{k}$ is called a *reuse iteration* of $\vec{j}$. Of particular interest is the last reuse access $\vec{a}$, and which is just the latest of reuse accesses among those that precede $\vec{a}$. Reuse vectors are used to succinctly capture last reuse access for references [104]. The reuse vector for R_a is $\vec{r_a} = (1, 0)$, and the reuse vector for reference R_b is $\vec{r_b} = (0, 1)$.

Two accesses have a **conflict relation** if they access different memory address, but the same cache block. In Figure 4.16, access $(2, 2, R_a)$ and $(3, 3, R_b)$ access different memory addresses, $a[2]$, and $b[3]$ respectively, but access the same cache block, $CB1$. Therefore accesses $(2, 2, R_a)$ and $(3, 3, R_b)$

[1] **Note:** The lexicographical size of a vector $\vec{v}$, denoted by $||\vec{v}||$ and simply called *size*, is defined as the number of points that are lexicographically less than $\vec{v}$ in the iteration space. Greater/smaller/minimum is also in the lexicographical sense.

conflict with each other. Finally, if both the memory addresses and the cache blocks of two accesses are different, then the two accesses are **unrelated**. In Figure 4.16, accesses $(1,0,R_a)$ and $(2,2,R_a)$ are unrelated, since they access different cache blocks.

In the absence of aliasing (e.g., the arrays are non-overlapping, the references are always accessed by their true names), there is no reuse between accesses of different references. However, there still may be conflicts between accesses of different references. Finally, if cache block size is equal to one element, there is just one reuse vector per reference to an array.

4.5 Cache Miss Equations

Any memory access that has a preceding reuse access but has no conflict access between itself and its last reuse access must result in a cache hit. Conversely, a conflict access between the two accesses to the same address will cause a cache miss in a direct-mapped cache. Thus in order to identify cache hits and misses, we need to know only two things: i) last reuse access, and ii) whether a conflicting access exists or not.

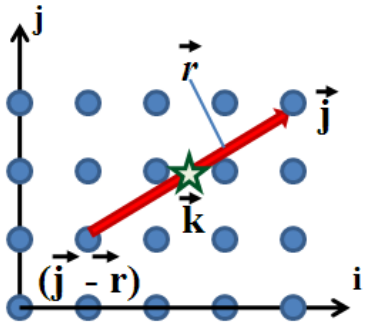


Figure 4.17: The access space of reference R is represented. Two successive accesses of elements of reference R ($(\vec{j},R)$ and $(\vec{j}-\vec{r},R)$) are labeled, and the reuse vector ($\vec{r}$) is marked by an arrow. A cache miss at $(\vec{k},S)$, between data accesses over the reuse vector $\vec{r}$ is marked by a star.

Finding conflicts among accesses is easy. We know that two accesses $(\vec{j},R)$, and $(\vec{k},S)$ conflict iff $CB(AF_R(j)) = CB(AF_S(k))$. Finding the last reuse access is a little tricky. First lets assume that there is only one reference to an array, and there is only one reuse vector per reference. Then for the reference R, we assume that $\vec{r}$ is the reuse vector, then by definition of reuse vectors, if some memory address is accessed in iteration $\vec{j}$, then it was last accessed in iteration $(\vec{j}-\vec{r})$, and both access the cache block $CB(AF_R(j))$. Now by our definition of cache miss, there will be a miss iff, some other reference, say S, accesses the same cache block in iterations $(\vec{j}-\vec{r})$ through $\vec{j}$. Figure 4.17 represents the cache miss over labeled points in the data access space. This is captured in the Cache Miss Equation:

$$CME_R^S(\vec{j},\vec{k},\vec{r}) := (CB(AF_R(\vec{j})) = CB(AF_S(\vec{k})))$$
$$\wedge((\vec{j}-\vec{r}) \prec \vec{k} \prec \vec{j}) \tag{4.2}$$

It states that the reference R will experience cache miss at iteration $\vec{j}$ along the reuse vector $\vec{r}$, due to another reference S in iteration $\vec{k}$, iff they access the same cache block. If the equality is satisfied for any value of $\vec{k}$, there is a cache miss at iteration $\vec{j}$. Now we can collect the iterations in which miss occurs:

$$MI_R^S(\vec{r}) = \{\vec{j} \in \mathscr{I} \mid \exists \vec{k} \in \mathscr{I}, CME_R^S(\vec{j},\vec{k},\vec{r})\} \tag{4.3}$$

MI_R^S is the set of all iterations $\vec{j}$ in which there is a cache miss for accesses of reference R due to a conflict with another reference S, along the reuse vector $\vec{r}$.

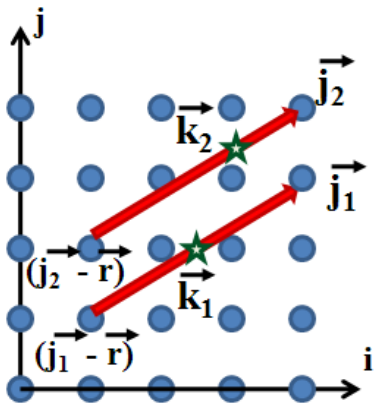


Figure 4.18: The access space of reference R, and the reuse of two data elements $((\vec{j}_1, a[1]), (\vec{j}_2, a[2]))$, by the same reuse vector, is represented. A cache miss occurs on the access $(\vec{j}_1, a[1])$ due to interference by $(\vec{k}_1, S)$. In the presence of multiple interfering references (with the same reuse vector), access $(\vec{k}_2, T)$ by reference T causes a cache miss at $(\vec{j}_2, a[2])$. The interfering accesses causing caches misses are marked by stars.

Till now we have only considered misses because of one other reference S. If there are multiple references, then there will be a miss at iteration $\vec{j}$, if there is a conflicting access due to any of the other references. Therefore,

$$MI_R(\vec{r}) = \bigcup_{S \in \mathscr{R}} MI_R^S(\vec{r}) \tag{4.4}$$

$MI_R(\vec{r})$ will be the set of all iterations $\vec{j}$ at which a cache miss occurs due to a conflict with *any* other reference (except another reference to the same array) by conflicting with the reuse due to reuse vector $\vec{r}$. This is depicted in Figure 4.18.

Now if there are multiple references to the same array, then it will result in multiple reuse vectors [34]. There can be more than one reuse vector, even if the cache block size is more than one

element. In that case, there is spatial reuse [33]. When there are multiple reuse vectors, then, a cache miss will occur at iteration $\vec{j}$, if there is a cache miss due to the smallest reuse vector. Noting that if there is a miss due to smallest reuse vector, then even the longer reuse vectors must suffer a cache miss, we can simply use the intersection operator. Therefore,

$$MI_R = \bigcap_i MI_R(\vec{r}_i) = \bigcap_i \Big(\bigcup_{S \in \mathscr{R}} MI_R^S(\vec{r}_i) \Big) \tag{4.5}$$

MI_R will contain the set of all iterations $\vec{j}$ at which a cache miss occurs for accesses of reference R due to any reuse vector, and any other reference. All the misses in the loop are then just a collection of misses of each reference.

4.6 Cache Vulnerability and Challenges in Estimation

Cache vulnerability(CV) is defined as the number of vulnerable bits in the cache, summed over the duration of a program execution, measured in byte-cycles. A bit is vulnerable if a soft error in it can destroy *architecturally correct execution* [75] of the processor. Any bit that is going to be overwritten is not vulnerable. Any bit in the data array that is protected with parity bit is not vulnerable if the cache block is clean and the bit is going to be accessed while the block remains clean. This is because a clean block can be simply invalidated if an error is detected in it. We assume, that all lines are protected by a parity bit, and therefore clean lines are not vulnerable.

Only cycle-accurate simulation based schemes are known for cache vulnerability estimation. While simulation based techniques are time consuming, they can be used in extremely embedded applications, where neither the program flow, or the data changes. However, if the data and its size can change, then simulation based techniques are of little help. In addition, the design space of some code transformations and data layout optimizations is so large, that exhaustive simulation is infeasible. Thus, except for in extremely embedded systems, an efficient technique to estimate data cache vulnerability is needed to decide on code transformations and data layout optimizations. An analytical model to estimate cache vulnerability offer the additional advantage of insights that we gain, and can then be utilized either apply these technique more to a different architecture/data set. In addition, it also provides a systematic and more informed mechanism to trade-off accuracy for the analysis time.

We build our cache vulnerability estimation technique similar to cache miss equations, but estimating cache vulnerability is far more complicated than cache misses. Cache miss equations estimate the number of cache misses, which is a subset of the cache accesses to the same data. In comparison, cache vulnerability is the sum of "time duration" between two consecutive accesses to the same data, when the second access is a read, and the data that was accessed was dirty. There are two main complications in

vulnerability estimation, as compared to estimating cache misses. These drive the key modifications and additions being incorporated in our cache vulnerability equation technique over that of the methodology in cache miss estimation:

1. Notion of "time" between accesses.
2. More information about the accesses, e.g., whether the access is a read or write, the knowledge of whether the data was "dirty" at the time of access.

While the second problem is simpler (in theory) and can be solved by adding more detailed information about references, the first is a fundamentally challenging problem. To compute cache misses, fundamentally for every access we only define a **Boolean function** $AM: \mathscr{A} \rightarrow \{0,1\}$ from the access space indicating whether there was a miss at the access. The misses in the program are then just specified as a subset of the access space, i.e., $Miss = \{\vec{a} \mid AM(\vec{a}) = 1, \vec{a} \in \mathscr{A}\}$. While enumerating the elements of $Miss$ is doubly exponential [21], the number of cache misses can be found in polynomial time by simply counting the number of elements in the set [33].

In contrast, to compute cache vulnerability, we need to define an **Integer function** $AV: \mathscr{A} \rightarrow \mathscr{Z}$ which captures the vulnerability of the data since it was last accessed. The program vulnerability can then be computed by adding the vulnerabilities of each access, i.e., $Vul = \sum_{\vec{a} \in \mathscr{A}} AV(\vec{a})$. One of the main challenges in computing cache misses is of converting the integer function into sets, such that the total vulnerability can be computed by finding the number of elements in a set.

Other practical challenges in vulnerability estimation is that analysis at iteration granularity is required, as compared to estimating cache misses in which analysis at cache access granularity suffices. Furthermore, since the dirty information in caches is maintained at a block level of granularity, a whole block is considered vulnerable if any single bit in it is vulnerable. This makes modeling cache vulnerability at word or byte granularity challenging. This is because, a word may be vulnerable even if there are no access to it at all – it can be vulnerable if the blocks containing them are dirty! Finally, even if we can exactly compute CV by considering all these factors, such a model is likely to be very complicated (as we will see in the paper), so as to jeopardize its practical use. Thus, an important challenge in vulnerability estimation is also to be able to make trade-off between modeling complexity and modeling accuracy, so that we can develop a relatively simple, yet accurate model of cache vulnerability.

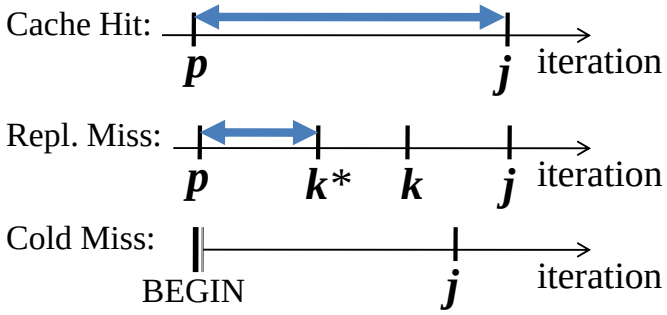


Figure 4.19: Access Vulnerability is the vulnerability from the last accumulated from the last access to the same data. **Cold Miss:** The vulnerable duration for the first access is 0. **Cache Hit:** The vulnerable duration is the length of the smallest reuse vector. **Cache Miss:** The vulnerable duration is the distance from the previous access to the first interfering access.

4.7 Cache Vulnerability Equations

Access Vulnerability

Unlike cache misses, which is an "event", vulnerability is computed as an "interval", or "duration". The key idea in computing vulnerability is to associate it with each access. The *Access Vulnerability*, $AV : \mathscr{A} \rightarrow \mathscr{Z}$ of an access $\vec{a} = (\vec{j}, R)$ is the vulnerability of the datum at the memory location $MF_R(\vec{j})$, since it was last accessed. If $\vec{a}$ is the first access to the data, then the datum is not considered vulnerable, or $AV(\vec{a}) = 0$. Similarly, if if this access is a write access, then it is not considered vulnerable. The reason is, that the datum is overwritten, and any error in it since the last access is inconsequential. Also if if the datum was not dirty at the time of access, then the datum is not considered vulnerable. This is because we assume that parity protection will detect the error, and the correct value can be read from the lower levels of memory, which we consider protected (through use of ECC or any other scheme).

The access vulnerability is non-zero only when the the access is a "read", and the datum was dirty at the time of access. However, the value of vulnerability depends on whether the access is a cache hit or a miss. If the access is a cache hit, then the datum was vulnerable for the whole duration from the last access to this access. Suppose that R is the only reference to the array, and it has only one reuse vector $\vec{r}$. Then the datum that is accessed by $\vec{a} = (\vec{j}, R)$ was last accessed by $\vec{b} = ((\vec{j} - \vec{r}), R)$. If the access $\vec{a} = (\vec{j}, R)$ is a cache hit, then the datum was vulnerable for the whole duration from $(\vec{j} - \vec{r})$ through $\vec{j}$, i.e., $AV(\vec{a}) = ||\vec{r}||$ (shown in Figure 4.19). However, if the access $\vec{a} = (\vec{j}, R)$ is a cache miss, then this datum was replaced by a conflicting access, say $\vec{c} = (\vec{k}, S)$, at iteration $\vec{k}$. Therefore, the datum was in the cache only from iteration $(\vec{j} - \vec{r})$ through $\vec{k}$. After this, from iteration $\vec{k}$ through $\vec{j}$, the datum was in the lower levels of memory, which we consider "protected". Therefore, $AV(\vec{a}) = ||\vec{k} - (\vec{j} - \vec{r})||$. However, even in the case when there are only two references, and only a single reuse vector per reference, the

other reference may access and conflict more than once between accesses $\vec{b}$ and $\vec{a}$. In this case, we must consider only the distance from the last access $(\vec{j}-\vec{r})$ to the first access that interferes $\vec{k^*} = min(\vec{k})$. All these cases are illustrated in Figure 4.19.

Representing Integer Function

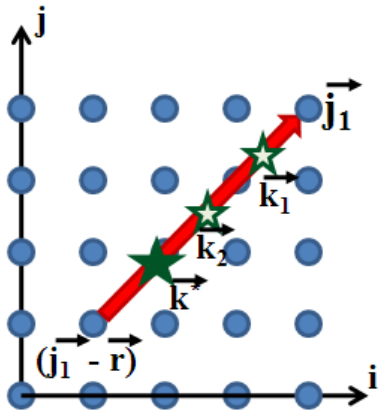


Figure 4.20: In the case of multiple references accessing the cache-block data, possible cache-misses can be recognized over multiple interference points over the reuse vector $(\vec{k}_1, \vec{k}_2, \cdots, \vec{k}^*)$. These are represented by the stars over the reuse vector. The first interference point For vulnerability calculation, we consider the first possible interference point $(\vec{k}^*)$, represented by the larger solid star.

A central problem in CV modeling is how to represent the integer function AV. Recall that AM is a Boolean function, which can be easily represented as a set. Our solution is to augment the vector $\vec{j}$ in Equation (4.3) with a scalar c, and let c take on all integer values less than the vulnerability l: $\{(\vec{j},c) \mid 0 \leq c < l = ||\vec{k}^*|| - ||\vec{p}||, \cdots\}$. Essentially we diversify each $\vec{j}$ exactly l times, so that we can get the total vulnerability simply by counting the elements of the set. However, still it is not obvious how to express $\vec{k}^*$. Since $\vec{k}^*$ is the earliest conflict iteration we would like to say $\vec{k}^* = \min\{\vec{k}\}$, or $\vec{k}^* \leq \vec{k}, \forall \vec{k}$. This phenomenon is represented by Figure 4.20. However, $\vec{k}$ is already qualified with existential quantifier $(\exists)$, and moreover adding universal quantifier $(\forall)$ causes the equation to be only a general *Presburger formula* and not a simpler Diophantine equation, greatly increasing the complexity.

We resolve this problem by counting *nonvulnerability* instead, i.e., the size of a reuse vector minus the vulnerability. Thus we first calculate *vulnerability capacity*, and subtract nonvulnerability from it to compute real vulnerability. This is depicted in Figure 4.21

Access vulnerability AV_r of reference r with only one reuse vector $\vec{v}$:

$$ANV_R^S(\vec{r}) = \{(\vec{j} \in \mathscr{I}, c) \mid \exists \vec{k} \in \mathscr{I},$$
$$0 \leq c < ||\vec{j}|| - ||\vec{k}||, \text{CME}_R^S(\vec{j},\vec{k},\vec{r})\} \tag{4.6}$$

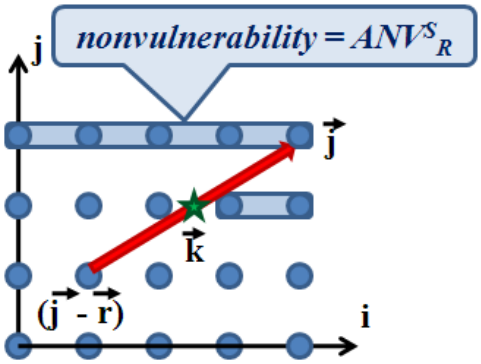


Figure 4.21: Access *nonvulnerability* = ANV_R^S), is the sum of iterations which are not vulnerable, between the two successive accesses on the reuse vector. The iterations after a cache-miss at $(\vec{k}, S)$, are not vulnerable and therefore are enclosed within a box. The remainder of the iterations, not enclosed within a box, between $(\vec{j} - \vec{r})$ and $(\vec{j})$ are computed using Equation (4.6).

$$ANV_R(\vec{r}) = \bigcup_S ANV_R^S(\vec{r}) \tag{4.7}$$

$$AV_R = ||\vec{r}|| \cdot |\mathscr{I}| - |ANV_R(\vec{r})| \tag{4.8}$$

Working with nonvulnerability also makes it easier to consider multiple references, as shown in Equation (4.7). Further, read-hit iterations are automatically taken care of in this formula; for a hit iteration $\vec{j}$, Equation (4.6) returns null since no $\vec{k}$ exists, and Equation (4.8) returns the correct value. However, cold miss iterations should be excluded, which is not done by the formula.

Multiple Reuse Vectors

The formula Equation (4.6)–Equation (4.8) has two limitations: incorrect handling of cold miss iterations, and considering only a single reuse vector. Those two limitations are closely related and can be solved at once by extending the concept of reuse vector with *domain*. In our formulation as well as in CME, reuse vectors serve the purpose of limiting the search space for conflict miss to the previous m iterations, where m is given by the size of a reuse vector. Ideally, m should be given by the last reuse iteration (LRI). However, exact computation of LRI is intractable in the general case, but most often it can be found from reuse vectors. A reuse vector $\vec{r}$ of reference R is derived from

$$MF_R(\vec{j} - \vec{r}) = MF_R(\vec{j}) \tag{4.9}$$

which suggests $\vec{j} - \vec{r}$ is a possible LRI of $\vec{j}$. In order for that to be the case, two conditions must be met: i) $\vec{r}$ should be valid on $\vec{j}$ for Equation (4.9), ii) there should be no smaller reuse vector valid on $\vec{j}$. We

call the set of iterations where a reuse vector is valid for Equation (4.9), the *domain* of the reuse vector. Certainly, domain can be defined for any reuse vector.

While there can be iterations that are not included in any domain (they are cold miss iterations), we can easily find the smallest reuse vector for any iteration that is included in at least one domain. Given a set of reuse vectors $\{\vec{r}_i\}$ sorted in their sizes, i.e., $\vec{r}_1 \prec \vec{r}_2 \prec \cdots$, and the corresponding set of domains $\{\mathscr{D}_i\}$, we define *differential domains*, $\{\mathscr{P}_i\}$, as follows.

$$\mathscr{P}_i = \mathscr{D}_i - \mathscr{D}_{i-1} - \cdots - \mathscr{D}_1 \tag{4.10}$$

Differential domain $\mathscr{P}_i$ is the set of iterations in which $\vec{r}_i$ is the smallest reuse vector. Clearly, differential domains are mutually disjoint, and do not include any cold miss iteration. Now we can easily extend Equation (4.6)–Equation (4.8).

$$\begin{aligned} ANV_R^S(\vec{r}_i, \mathscr{P}_i) = \{(\vec{j} \in \mathscr{P}_i, c) \mid \exists \vec{k} \in \mathscr{I}, \exists n \in \mathscr{Z}, \\ 0 \le c < ||\vec{j}|| - ||\vec{k}||, \mathrm{CME}_R^S(\vec{j}, \vec{k}, \vec{r}_i, n)\} \end{aligned} \tag{4.11}$$

$$ANV_R(\vec{r}_i, \mathscr{P}_i) = \bigcup_S ANV_R^S(\vec{r}_i, \mathscr{P}_i) \tag{4.12}$$

$$AV_R = \sum_i \left(||\vec{r}_i|| \cdot |\mathscr{P}_i| - |ANV_R(\vec{r}_i, \mathscr{P}_i)| \right) \tag{4.13}$$

Access Type and Cache block State

So far we have considered how to model the effect of cache hit/miss on CV. Now we consider how to model the effect of read vs. write difference. AV_R in Equation (4.13) is accurate for read accesses. For write accesses we need to exclude vulnerability due to hit accesses, which we call *hit-nonvulnerability*. Hit-nonvulnerability, HNV_R:

$$HNV_R = \sum_i ||\vec{r}_i|| \cdot |HI_R(\vec{r}_i, \mathscr{P}_i)| \tag{4.14}$$

$$HI_R(\vec{r}_i, \mathscr{P}_i) = \mathscr{P}_i - MI_R(\vec{r}_i) \tag{4.15}$$

where $MI_R(\vec{v}_i)$ is calculated by Equation (4.3)–Equation (4.4).

Modeling cache block state is less obvious than access type. Exact modeling requires looking even beyond LRI ($= \vec{j} - \vec{r}$) for any write to the same memory block, with the search space expanded up to the next conflict access. Compared to the formulation developed so far, which needs to find only one conflict iteration ($\vec{k}$), this new modeling requires two more (write reuse, the next conflict), which

will greatly impact the complexity. Fortunately, for loops with uniformly-generated references we have a much simpler rule. Among uniformly-generated references we can define a total order from their leading/trailing relationship. For instance, in a *i*-*j* loop nest, $A[i][j]$ trails $A[i+2][j+3]$ at the distance of $[2,3]$, which is in fact one of the reuse vectors of $A[i][j]$. We consider that a reference accesses only clean blocks if it does not follow a write reference. The other references are considered to access dirty blocks. We understand that this simple rule is only an approximation and is not always correct. However, it can be very easily applied and yet highly accurate if a write reference has no group reuse or the group reuse vector is small, which is the case in many loops including matrix multiplication.

Post-access Vulnerability

Our access vulnerability can account for only the portion of CV that becomes certain by the last access to each memory block. After the last access, there can be no more reuse but only zero or more conflict accesses. If a conflict access exists, the vulnerable interval extends to the first conflict access; otherwise, the vulnerable interval extends to the end of the program (provided that the block is dirty). Thus we need to find out i) the set of iterations in which the last accesses (per memory block) are made, and ii) the lengths of vulnerable intervals.

First, given a reference R, the set $\mathscr{P}_*$ of iterations for last accesses can be found from the *ranges*, $\{\mathscr{R}_i\}$, of reuse vectors, $\{\vec{r}_i\}$. Range is defined similarly to domain except that $-\vec{v}$ is replaced with $+\vec{v}$ in Equation (4.9). Then it follows from the definition that $\mathscr{P}_*$ is the set of iterations that are not included in any range, or $\mathscr{P}_* = \mathscr{I} - \mathscr{R}_1 - \mathscr{R}_2 - \cdots$. Second, the vulnerable interval is either $|\mathscr{I}| - ||\vec{j}||$ or $||\vec{k}^*|| - ||\vec{j}||$, whichever is the smaller, where $\vec{k}^*$ is the earliest of, if any, future conflict iterations. Again we use nonvulnerability to find this interval. Post-access vulnerability of reference R is $PV_R = |U_R| - |PNV_R|$, where $U_R = \{(\vec{j} \in \mathscr{P}_*, c) \mid 0 \le c < |\mathscr{I}| - ||\vec{j}||\}$ and PNV_R, the post-access nonvulnerability, is the union of all the post-access nonvulnerabilities PNV_R^S from different references S. Finally, $PNV_R^S = \{(\vec{j} \in \mathscr{P}_*, c) \mid 0 \le c < |\mathscr{I}| - ||\vec{k}||, \exists \vec{k} \in \mathscr{I}, \exists n \in \mathscr{Z}, \text{CME'}_R^S(\vec{j}, \vec{k}, n)\}$, where CME′ is CME with $(\vec{j}, R) \prec (\vec{k}, S)$ substituting for the original range constraint.

Implementation and Complexity

Algorithm 1 lists the procedure to compute access vulnerability for all references in a loop (post-access vulnerability can be computed similarly). The core of this procedure is writing extended CMEs (line 7) and counting the integer points in them (line 9). CME, extended or not, is a set of constraints that specify a polytope possibly using existential quantifier, and counting integer points in such a polytope can be done in polynomial time using the *barvinok* library [102], which is based on *PolyLib* [65]. However, further complication comes from the union operation in between (line 9), which exists in CME as well.

Algorithm 1: Find access vulnerability of all references

1: **for all** $R \in \mathscr{R}$ that can access dirty cache blocks **do**
2: integer (vulnerability of R): $V_R \leftarrow 0$
3: Find all the reuse vectors $\vec{r}_i$ and their domains $\mathscr{D}_i$
4: **for all** i in the increasing order of $||\vec{r}_i||$ **do**
5: Find $\mathscr{P}_i$ from $\{\mathscr{D}_i\}$
6: **for all** $S \in \mathscr{R}$ **do**
7: Find $ANV_R^S(\vec{r}_i, \mathscr{P}_i)$ /* CME extended for CV */
8: **end for**
9: integer: $ANV_R(i) \leftarrow |\cup_S ANV_R^S(\vec{r}_i, \mathscr{P}_i)|$
10: $V_R \leftarrow V_R + ||\vec{r}_i|| \cdot |\mathscr{P}_i| - ANV_R(i)$
11: **end for**
12: **if** R is a write reference **then**
13: Compute HNV_R using CME
14: $V_R \leftarrow V_R - HNV_R$
15: **end if**
16: **end for**

There are several ways to handle unions. A simple method is to convert unions into intersections (intersections pose no problem) using the inclusion-exclusion property ($|A \cup B| = |A| + |B| - |A \cap B|$), which has unfortunately an exponential complexity. Another way is to use Pugh's method of converting unions into disjoint unions [88]. Counting the integer points is repeated for each reuse vector (line 4) and for each reference that can access dirty cache blocks (line 1). In our current implementation the complexity is dominated by the handling of union operator, and is $O(c \cdot N \cdot 2^{|\mathscr{R}|})$, where c is the average time for handling unions, and N is the total number of reuse vectors of references that can access dirty blocks.

4.8 CVE Model Validation

To validate our static analysis as well as to demonstrate its effectiveness and usefulness in program optimization we use the matrix multiply loop kernel. Our static analysis is performed using an automated analysis flow, which first derives reuse vectors and their domains from application description, then generates vulnerability equations, and finally calculates cache vulnerability using an integer-point counting engine. Simulation is performed using the SimpleScalar cycle-accurate simulator [14]. In all our experiments we assume that the L1 data cache is write-back and direct-mapped, with 32-byte line size. The cache size is set to 1$\sim$4 KBytes depending on the application's memory footprint. Small cache sizes are chosen not only to model embedded systems but also to induce frequent cache misses, which will create more variety in the number of cache misses and cache vulnerability, and thus make it more challenging to predict the cache behavior.

Loop interchange is a well-known loop optimization that changes the order of loops in a loop nest. Since it can completely reorder the memory accesses in a loop, loop interchange can greatly affect cache vulnerability as well as cache misses. As we will see in our experimental results, there is usually

much greater variation in cache vulnerability than in the number of cache misses. Moreover, the loop order with the least number of cache misses is not always the one with the lowest cache vulnerability. This suggests that in order to address reliability issues, compilers should specifically target vulnerability reduction rather than just cache miss reduction.

Loop	Analytical			Simulation			
order	CV(li)	#CM	ACV	CV(li)	#CM	CV(bc)	RT(c)
ikj†	2071	538	**1321**	2071	538	**1.71M**	**41.2K**
kij	5488	788	2874	5488	788	4.67M	45.9K
ijk*	6744	418	2669	6744	**418**	**5.07M**	**39.0K**
kji	15163	1746	7377	15163	1746	16.71M	68.5K
jik	33852	598	8816	33852	598	22.59M	42.5K
jki	32341	1544	11732	32341	1544	33.06M	65.4K
Corr.	1.000	1.000	.995	–			

Table 4.1: **Vulnerability Results for `mmult`, N=12** Legend – CV: cache vulnerability, CM: cache miss, ACV: adjusted CV, RT: runtime, (li): line-iteration, (bc): byte-cycle, and (c): cycle.

Table 4.1 compares the cache vulnerability of `mmult` (matrix multiplication) computed by our static analysis and by simulation. The adjusted cache vulnerability (ACV) is calculated using the number of cache misses (#CM) predicted by the CM equations [33] with modifications due to domains. The rows are sorted in the increasing order of simulation CV in byte-cycles (7th column). The last row lists the correlation coefficient between each column on the analytical side and the corresponding column on the simulation side, with ACV corresponding to CV in byte-cycles on the simulation side.

First, we can see that the second column (analytical CV in line-iterations) exactly matches the fifth column (CV in line-iteration from simulation). `mmult` has nontrivial access pattern in that all three references have different pairs of spatial and temporal reuse vectors. Thus this validation result gives some assurance of our CV equations. In the table, we also observe that the number of cache misses predicted by the CM equations, with modifications due to domains, is 100% accurate as compared to simulation. Finally, the ACV numbers also closely follow the simulation results, with a very high correlation.

In addition to the basic validation results, there are interesting points to observe from the table. First, the CV variation is much higher than CM variation or RT (runtime) variation. Cache vulnerability, as measured in byte-cycles from simulation, varies from 1.71M to 33.06M, or more than 19 times, whereas the number of cache misses and runtime vary by mere 3.2X and 1.7X respectively. Therefore the effect of compiler optimizations for cache vulnerability can be greater than for cache misses. Second, the loop order for the minimum RT is not the same as the one for the minimum CV. The original loop order, which is marked with an asterisk in the first column, has minimum CM and consequently minimum

runtime as well. However, if we choose another loop order, marked with a dagger, the cache vulnerability can be reduced by almost three times while the runtime is increased by only 5.7% (The minimum-CV loop-order can be correctly predicted by our analysis as shown in the table). Please note that the cache vulnerability in byte-cycles already takes into account the effect of increased runtime; therefore, the three times reduction in CV is the real reduction that we can expect to see in the soft error rate of the data array of L1 data cache. The above two points strongly suggest the need and scope of compiler optimizations to reduce cache vulnerability, which has been neglected in traditional loop optimizations focusing on cache misses only. Our static analysis can be an important first step toward compiler optimizations for cache reliability.

Loop	Analytical			Simulation			
order	CV(li)	#CM	ACV	CV(li)	#CM	CV(bc)	RT(c)
ikj†	3622	2173	**3055**	3622	2333	**3.19M**	**92.3K**
ijk*	9892	1462	5230	10778	**1574**	**9.25M**	**79.4K**
kij	10221	2653	6520	8988	2773	10.47M	99.1K
kji	26150	3658	13575	26103	3801	35.79M	130.3K
jik	63882	1564	18549	53568	1692	49.15M	82.9K
jki	66581	3438	24793	57827	3565	81.88M	127.1K
Corr.	.9978	.9998	.9919	–			

Table 4.2: **Vulnerability results for `mmult`, N=14** Legend – CV: cache vulnerability, CM: cache miss, ACV: adjusted CV, RT: runtime, (li): line-iteration, (bc): byte-cycle, and (c): cycle.

A potential weakness of our technique, as it relies on reuse vectors to simplify the equations, is the inaccuracy of reuse vectors and their domains. The prediction of reuse vectors on the last reuse access can become less accurate at the boundary of the iteration space. In our first example where $N = 12$, the iteration space is divided by the cache line size in all its dimensions (a cache line contains exactly four array elements of double word each). If we change N to 14, the boundary effect starts to appear, which is shown in Table 4.2. In the table we notice that even the CM equations start to disagree with simulation although the overall correlation is very high. Comparing Columns 2 and 5 (CV in line-iterations), our CV equations tend to be more accurate in low CV region while it amplifies in high CV region. Our CV analysis sometimes loses on the details but it accurately captures the overall trend. Most importantly, the ordering in the adjusted CV exactly matches the ordering in the simulation CV in byte-cycles. Again in this example, we observe the same pattern that the loop order for minimum CM is different from that of minimum CV, and there is much more to gain in terms of cache vulnerability if we can make a little trade-off in terms of runtime.

```
for ( i = 0 ; i < N ; i++ )
  for ( j = 0 ; j < N ; j++ ) {
    A_{j,i+1} = f1(P_{j,i+1}, P_{j,i}, U_{j,i+1})
    B_{j+1,i} = f2(P_{j+1,i}, P_{j,i}, V_{j+1,i})
    C_{j+1,i+1} = f3(V_{j+1,i+1}, V_{j+1,i}, U_{j+1,i+1},
                     U_{j,i+1}, P_{j,i}, P_{j,i+1}, P_{j+1,i+1}, P_{j+1,i})
    D_{j,i} = f4(P_{j,i}, U_{j,i+1}, U_{j,i}, V_{j+1,i}, V_{j,i})
  }
```

Figure 4.22: Calc1 loop from swim (after loop interchange).

4.9 Analytical Optimization of CVE: Case Study

Many loop transformations significantly affect cache vulnerability, often much more than cache misses. While our cache vulnerability equations can be used to accurately compute the total cache vulnerability of a loop nest, and thus can guide compiler optimizations, evaluating the equations is not always easy due to the limitations of back-end tools. Here we showcase alternative use cases of our cache vulnerability equations, using data placement.

Array Placement

In loops, array placement can dramatically affect the number of cache misses and cache vulnerability. There are two ways to change array placement. Intra-variable padding increases row sizes to reduce cache conflicts (both self and cross), which increases memory footprint. Inter-variable padding, or array placement adds unused space between arrays, or changes the base addresses of arrays, to reduce cache conflicts between different arrays. We use array placement to demonstrate the effectiveness of analytical optimization on cache vulnerability.

Figure 4.22 shows an abstract version of a loop nest from swim after loop interchange (detail is omitted to avoid copyright infringement). Hereafter we refer to the loop-interchanged loop as the original loop. This loop involves 7 arrays and many more references with very complex access patterns. Exhaustive exploration of array placement parameters for such a loop is prohibitive. For a very small 1KB cache, and even after restricting the base addresses to the cache line boundary (=32B), the design space has still $(2^5)^6 = 2^{30}$ combinations. Instead, we can quickly find optimal points exploiting the intuition provided by our CV equations.

Our CV equation has two parts. It first computes the total vulnerability *capacity* and then subtracts nonvulnerability from it. Often the total vulnerability capacity is not affected by base addresses. Therefore our goal is to maximize nonvulnerability by changing base addresses. Nonvulnerability is proportional to the distance between the current iteration and the earliest conflict iteration after the previous

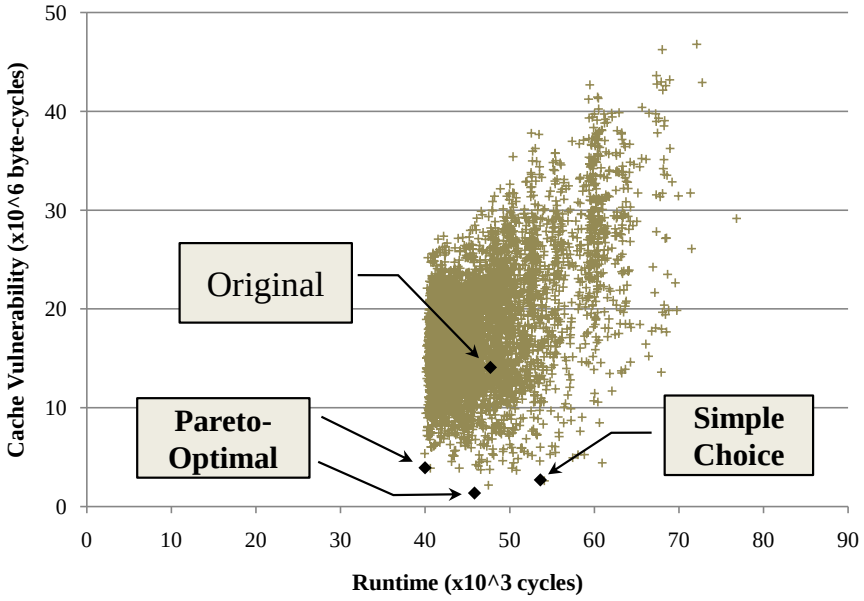


Figure 4.23: Cache vulnerability and runtime reduction through array placement.

reuse. In other words, maximum nonvulnerability occurs if a dirty cache line is evicted immediately after it is accessed, which agrees with the intuition. However, since frequent cache conflict will increase runtime, which negatively impacts cache vulnerability, our strategy is to evict as soon as possible those dirty cache lines that will *not* be accessed for a long time.

In our original loop in Figure 4.22, dirty cache lines are generated only by the four LHS (left-hand side) references. We can evict those lines by creating conflicts between each of them and any of the ensuing accesses. Let us use write accesses for that, since write misses generally incur less penalty. Then we have a chain of conflicts like this: $A_{j,i+1} \rightarrow B_{j+1,i} \rightarrow C_{j+1,i+1} \rightarrow D_{j,i}$. The last reference $D_{j,i}$ can be made to have conflict with one of the read references in the next iteration.

To derive formulas let us assume that all the arrays are initially placed at offset zero modulo the cache size, and that we can independently control the offset μ_X of each array X. Note that this can be implemented very easily without losing optimality. Let us also assume $\mu_A = 0$. Then the above chain of conflicts gives the offsets of the other three arrays. For instance, between A and B:

$$Addr(A_{j,i+1}) \equiv Addr(B_{j+1,i})$$
$$\Leftrightarrow \quad \mu_A + jM + (i+1) \equiv \mu_B + (j+1)M + i$$

where $\equiv$ is equality under modulo on the cache size and M is the size of each row (assuming every array has the same row size).

For the last reference, we must consider the next iteration, which can be either the next j-iteration or the next i-iteration. For each case we can set up a different array to have conflict with $D_{j,i}$. We explore two choices.

(1) Simple choice: Using $U_{j,i+1}$ and $P_{j,i}$

$$Addr(D_{j,i}) \equiv Addr(U_{j,i+1})|_{j=j+1}$$
$$\Leftrightarrow \quad \mu_D + jM + i \equiv \mu_U + (j+1)M + (i+1)$$

and

$$Addr(D_{j,i})|_{j=N-1} \equiv Addr(P_{j,i})|_{i=i+1,j=0}$$
$$\Leftrightarrow \quad \mu_D + (N-1)M + i \equiv \mu_P + (i+2)$$

Figure 4.23 plots CV and runtime results from simulation for random offsets (5000 instances). It also shows the CV and runtime for the original loop, which is about the center of the distribution. For parameters we use $N = 14$ and $M = 16$, and the cache size is set to 1KB. Compared to the original loop, our simple choice can reduce CV by more than 80%, which further validates our static vulnerability model. However the runtime is significantly increased. Although it is not surprising given that we have tried only to reduce CV, it suggests that cache misses should be considered in order to get truly optimal parameters.

(2) Pareto-optimal: To contain the runtime increase problem we resort to traditional CM reduction methods such as [33]. The key idea is to make the read references conflict as little as possible. Close examination of the offsets determined by the simple choice reveals that $\mu_U = \mu_C$ and $\mu_V = \mu_A$, which creates unnecessary conflicts and increases runtime. The latter is because we did not set any constraint on μ_V, which defaulted to zero, and the former is by chance. To improve the situation we set up a different reference $V_{j+1,i}$ to have conflict with $D_{j,i}$ along the j-loop, and used either $P_{j,i+1}$ or $P_{j+1,i}$ to have conflict along the i-loop, which gives two sets of parameters. Then the remaining free array U is assigned an offset that is farthest away from all the other arrays. The simulation results for these sets of parameters are shown in Figure 4.23 (marked as Optimal). Both points are pareto-optimal, and reduces CV by up to 90% or runtime by up to 16% compared to the original loop. Table 4.9 summarizes the exploration results.

4.10 Related Work

Solutions to mitigate the impact of soft errors are being sought after at all levels of computer design e.g., careful selection and screening of materials [9], SOI fabrication technologies [16], increasing the

Loop	CV (bc)	%reduc.	RT (c)	%incr.
Original (loop-interchanged)	14.08M	–	47.7K	–
Simple choice	2.69M	80.9%	53.6K	12.4%
Pareto-optimal 1	1.36M	90.3%	45.8K	–3.9%
Pareto-optimal 2	3.92M	72.1%	40.0K	–16.2%

Table 4.3: Array placement optimization results for swim

transistor size, adding passive capacitance, or changing the transistor types with threshold voltage shifts, adding gated resistors [91], partially protected caches [60], software duplication [89], to triple modular redundancy [86].

As opposed to hardware techniques, software techniques reserve the advantages of *flexibility* of application, and therefore the overheads thereof. Indeed, the most important benefit of software schemes is as a last-minute fix. For example, if it is required to improve the system reliability after system design, then only software techniques may be easily applicable. Most existing software approaches that attempt to improve reliability and mitigate the effect of soft errors are based on some form of program duplication [23, 35, 79, 89], and therefore incur severe power and resource overhead. This work is fundamentally different from all those previous software approaches – we study code transformations that will improve the reliability of application programs – without re-executing any instruction of the program. Therefore our techniques can have much less power, performance, and resource overheads.

Caches are one of the most vulnerable microarchitectural components in the processor and several techniques have been developed to reduce failures due to soft errors in caches, e.g. [12, 60, 76, 97], however the effect of code transformations, and in general compilers has not been evaluated. While there has been recent work in developing compiler techniques for register file protection [57, 105], there are no compiler approaches to mitigate the impact of soft errors in caches. Vulnerability [75] is the measure of failure rate of caches, and only simulation-based techniques are known to estimate it [97]. The ability to estimate the vulnerability for any given code is fundamental to not only driving, but even developing any compilation technique to optimize for vulnerability, and in general, simulation based techniques are not usable. This underscores the need for more efficient techniques to estimate vulnerability of data in caches.

This dissertation proposes a static analysis to estimate program analysis, and our approach builds upon cache miss analysis [33]. While there is a more general approach [21] to model reuses in Presburger arithmetic [87], we use the reuse-vector based approach [104], since it is much more tractable. We use the Omega library [87] to perform polygon union and intersection operations and Polylib [65] to count the number of points in the polygons containing vulnerable iterations.

4.11 Summary

To combat the threat of soft errors, techniques have been developed at all abstractions of processor design. Software schemes are particularly useful since they provide flexibility of application and therefore overheads, and can be applied in current or even previous generation processors, and most importantly, are irreplaceable as a last-minute fix. Caches are the most "vulnerable" component in the processor, and traditional ECC-based techniques are getting used up in manufacturing errors, and effectively only parity protection remains. For the first time, in this work, we perform design space exploration to determine the impact of algorithm choice, and system I/0 data-set sizes, on the vulnerability of a system. We also perform experiments to analyze the impact of various system design choices towards improving reliability with limited performance trade-off. Our experiments thus reveal the need for an efficient and accurate means to estimate data cache vulnerability; while considering both the program data access behavior and also the cache microarchitecture configuration – for which only simulation based techniques are known. This paper develops for the first time, analytical techniques to efficiently and statically estimate data cache vulnerability of programs; opening the doors for compiler techniques to make trade-off decisions between: power and performance for reliability. In addition, we demonstrate how the insights from vulnerability calculations can be used to innovate simple practical schemes to reduce program vulnerabilities.

Chapter 5

HYBRID COMPILER MICRO-ARCHITECTURE TECHNIQUES: FOR RELIABILITY

"To improve system reliability, researchers have developed techniques at the hardware micro-architecture level [45, 62, 91] and compiler level [57, 60]. By integrating the system information captured from the compiler and micro-archtiecture design layers, the advantages and accesible resources at each design layer can be best utilized for improved system benefit. The impact of soft errors in a system is a completely random phenomenon. Over the years, in an attempt to provide reliable computing, researchers have been limited by the overheads (power and performance) in enabling protection throughout the duration of system execution; though the occurrence of soft errors is rare and unpredictable."

In this chapter, we highlight on the the trade-off between system reliability, and additional power consumption for protection. We propose a smart mechanism that extracts information through program analysis, and then enforces the hardware architecture with this information, to efficiently provide *power-efficient reliability*.

5.1 Introduction

To model the susceptibility of data in caches, the metric of *vulnerability* is used. A datum is vulnerable (or susceptible to data corruption by soft errors) in the cache, only if it is dirty (written by the processor), *and* is then either, i) read by the processor, or ii) written back to the next level of memory. Herein, the assumptions of the underlying cache architecture are that:

i) the probability of double-bit errors is negligible (typically 3 orders of magnitude lesser [76]) than single-bit errors; simple hardware techniques like interleaving the bits of a cache-line can reduce the onset of multi-bit errors.

ii) data in the cache is protected by parity bit error detection [38] (as in popular processor architectures like Intel Xscale® [48], Intel IA-32® [49], AMD Athlon® [3], etc.).

In a cache where every cache-line is protected by parity bits, if an error is detected on a cache-line that is not dirty (.i.e., clean or not updated), it can be invalidated and re-loaded from the memory as a cache-miss. One method to protect cache data from soft errors, is by ensuring that an updated copy of all the cache-data is available in the memory (to re-load, when an error is detected). A write-through cache ensures such a scenario, by writing copies of cache-blocks as and when they are updated in the cache, thereby realizing *zero* vulnerability. However, write-through caches suffer from very high memory traffic between the cache and the rest of the memory subsystem. These memory-writes keep the data-bus busy,

thereby increasing the cache-miss latency for new memory accesses, and affect the overall performance of the system. Another consequence is excess energy consumed by the memory subsystem:

i) at the data-bus between the cache and the lower levels of memory (which are typically off-chip in embedded systems), and

ii) by accesses to the lower level memory components on every write-back; increasing the total power consumption of the system.

To find a middle ground, Early Write-Back (EWB)[63] cache architecture was proposed; In this, all the dirty cache-blocks are written back to the next level of memory only at periodic intervals. Reducing the frequency of write-backs, reduces the memory traffic and therefore the power consumption of the system, but at the cost of cache-data vulnerability, when compared to a write-through cache. By varying the periodicity of cache write-backs, EWB caches can explore the inversely proportional trade-off between vulnerability reduction and power overhead due to the additional memory traffic.

Both these techniques write-through (WT), and early write-back (EWB) are hardware techniques, and are not sensitive to the data access patterns of the application. For example, if a datum is vulnerable across two write-back periods (in the EWB technique, by executing write-backs at designated intervals), the additional write-backs eventually do not affect the vulnerability realized, but only increases memory traffic. If the cache write-back process can be customized according to the changing data access pattern of applications, vulnerability reduction can still be achieved, but with reduced number of write-backs, and thus reduced power overheads. Thus, there is scope for power-efficient vulnerability reduction by customizing the write-backs based on the data access patterns of data in a program.

In this chapter, we propose a hardware-software hybrid scheme: Smart Cache Cleaning (SCC), that provides a means to dynamically moderate the cache write-backs and thus achieve power-efficient vulnerability reduction in embedded systems. Our scheme is composed of three important components:

1. application analysis to determine, *which data accessed in the program, has to be write-back and when the said write-back has to be executed*, to achieve power-efficient vulnerability reduction,
2. *succinctly represent the time sequence identified* as to when a reference must be written back, and
3. transfer this information to the *specialized SCC architecture*, that performs write-backs of the specified references at specified times.

Our experiments over scientific benchmark loops like Livermore[4] and LINPACK[70] show that smart cache cleaning achieves 26% better energy-vulnerability product than the lowest energy-vulnerability product achieved by the EWB scheme (across various write-back periods). Our SCC scheme achieves almost *zero* vulnerability, at $< 1\%$ power overhead. Using the EWB scheme, to achieve the same level of vulnerability, a minimum of 40% and an average of $2.88\times$ power-overhead is incurred.

5.2 Background: Intermediate Vulnerability Time (IVT)

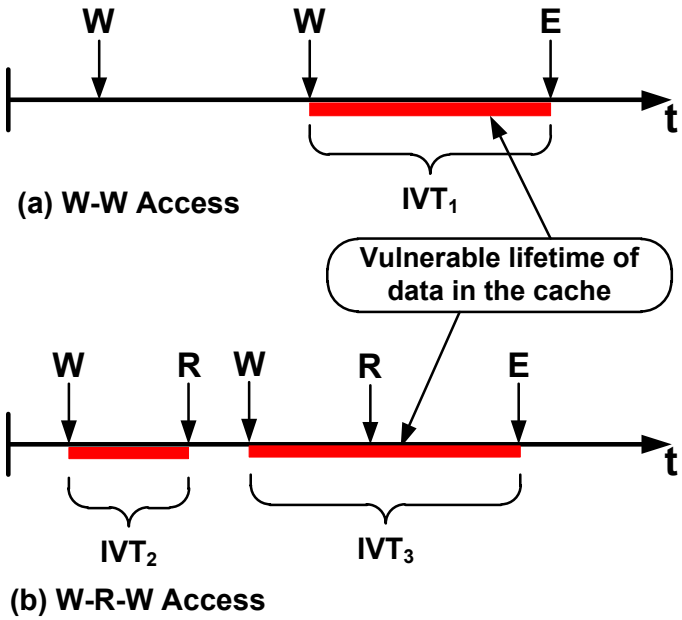


Figure 5.1: Intermediate Vulnerable Time (IVT) .i.e., the time a data element remains vulnerable in the cache is defined for two data access patterns (where, W=Write, R=Read, E=Eviction): a data element once written is vulnerable as long as it remains in the cache across read accesses. Since the second write (W) operation over-writes the updated data element, any error that may affect the unused data in the cache, deems the access as *not-vulnerable* for that time slot.

On a store operation from the processor, a data element in the cache (byte or word) is written. The containing cache-line now is deemed dirty, such that this becomes the only updated copy of the data stored. The time that such dirty cache-data remain in the cache, it is vulnerable to data corruption by soft errors. Over the sequence of different accesses on the cache-data (write (W), read (R), eviction or write-back (E)), the vulnerability of the data varies based on its usage. We define *Intermediate Vulnerable Time (IVT)*, as the duration for which a data element is vulnerable in the cache; after being updated by a write operation. Data access patterns in a program can be broadly classified into two types WW (write only) and WR (write and read). In Figure 5.1, the two data access patterns over a data element are portrayed, and the IVT definition in each case is highlighted:

a) The updated (written) data may be over-written by the program. In this case, since the updated data (in the first write operation) is not used but again updated, any soft error on the stored data between the two write accesses is not recognized. Therefore, the IVT for the data here is only the time from the last write operation to the time it was evicted (E) from the cache; when the data updates the underlying memory.

b) The updated (written) data may be used by other data accesses (read) during the course of the program. Since the correctness of this data is essential for the correct functioning of the system, it is vulnerable throughout this duration in the cache (till eviction E). However, as in Figure 5.1(b), if the same data is updated by another write operation, the data is over-written; therefore, the time slot between the last use (read) and update (write) operation is deemed *not-vulnerable*.

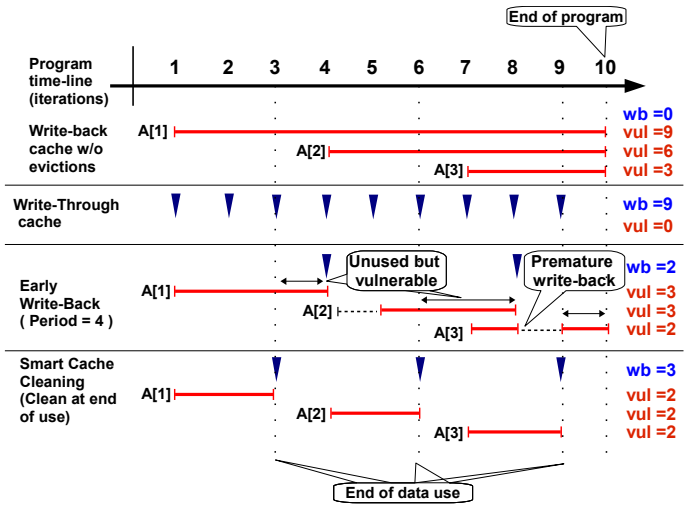
5.3 Motivation

```
for(i : 1 to 3){
  for(j : 1 to 3){
    A[i]+= B[j];
  }
}
```

(a) Example program

Cleaning Period	Vulnerability of Array A	Additional write-backs
No WB	18	0
WT	0	9
EWB 4	8	2
SCC	**6**	**3**

(b) Tabulated program statistics



(c) Cache write-backs and its vulnerability impact

Figure 5.2: Demonstrating the need and importance of smart cache cleaning. (a) Two dimensional loop operating over two data arrays one read-only (B) and the other read-and-written (A). (b) Summary of array A's vulnerability in each cache configuration shows the SCC scheme achieves energy efficient vulnerability reduction. (c) Detailed iteration-level analysis of cache write-backs, in each configuration, and their impact on the vulnerability of array A's elements.

We motivate the need for a mechanism to dynamically moderate the cache write-backs, with help of an example as in Figure 5.2(a). For this, we take a two-dimensional loop operating over two arrays, executing for a total of nine iterations. We assume that, during the course of the program, there occur no cache-evictions or write-backs due to cache-line replacements; the program terminates on the 10^{th} iteration or cycle and all the cache-data is finally written-back to the memory. We again assume that the cache is protected with a parity-bit error detection mechanism, such that its copy from the memory can be re-loaded, if an error is detected on the cache-data. From the definition of *vulnerability*, we

observe that the three elements of array `B` are only read, and therefore not vulnerable. On the other hand, the three elements of array `A` are updated (read and written) on every iteration, and therefore vulnerable for the entire time that they remain in the cache. In Figure 5.2(c), a time-line for the program execution is drawn and below it, in each section is the cache behavior on the elements of array `A`, in each cache write-back configuration. Immediately below the time-line in Figure 5.2(c), the position of each element of array `A` denotes the time it is first accessed by the program (for e.g., `A[2]` is first accessed when index `i=2` and `j=1` on the 4^{th} iteration of the program); the vertical dotted line that follows, indicates the last iteration it is updated/used by the program.

In the baseline write-back cache configuration, once an element is loaded into the cache, it is not evicted by any additional write-backs. Therefore the data element remains vulnerable from its first access (write access) till the end of the program; described by red bars, for each element, extending from the start to end of its life-time in the cache. The total number of write-backs (wb) and the vulnerability (vul) of each data element is marked on the right of the time-line. In the write-through cache configuration, on each iteration when the data in the cache is updated, it is also written-back to the memory. The downward pointing arrows in Figure 5.2(c) denote the write-back operations in the cache, on every iteration of the program; which is a characteristic of the write-through cache configuration. Rightly so, the vulnerability of data in the cache is 0, because there always exists a copy of the updated data in the lower level memory (which can be re-loaded when an error is detected in the cache), and data in the cache is always *clean*.

On similar lines, we explore the vulnerability vs write-back count ratio of the early write-back (EWB) cache architecture and our customized smart cache cleaning scheme. Li et al. [63], report through design space exploration the power efficiency trade-offs involved in the choice of a write-back period. Based on their recommendations and using a conservative estimate for the sample program and cache model considered, we set the write-back period to be 4 iterations. In this, once every 4 iterations of the program, the EWB architecture identifies *dirty* data in the cache and writes-back the same into the lower-level memory, thus rendering the data *clean*. We observe that this mechanism achieves 55% reduction in data-cache vulnerability, at the cost of only 22% additional write-backs (compared to the WT cache)to the lower-level memory. The highly regular nature of the sample program here, ensures such a profit by this technique, but such is not the case in general purpose applications. We arrive at such a conclusion owing to some key observations on the operation of the EWB scheme:

1. The *pre-defined periodic nature of the write-backs* rarely corroborate with the data access patterns of the application. Cache-data here, more often than not, tend to remain vulnerable beyond the time

they are required and/or updated by the program. For example, in Figure 5.2(c) the double-ended arrows indicate the time that each data element remained vulnerable, after it was last updated by the program. This functionality of the EWB scheme, causes the array `A` to be, vulnerable for an additional 4 unused iterations.

2. The working of the EWB scheme is to *identify all dirty cache-lines on each period and perform write-backs on all of them*, may require writing-back (or cleaning) data when it can be used/updated by the program in the immediate future. For example, in Figure 5.2(c), during the access time of `A[3]`, a periodic write-back (once every 4 iterations) cleans `A[3]` in addition to the previously accessed `A[2]`. However, since `A[3]` is updated the very next iteration (which in-turn is the last iteration it is used), the data remains vulnerable from then till the end of the program. This functionality of the EWB scheme, while reducing vulnerability by *one* iteration, causes the data to remain vulnerable for *one* additional iteration; the additional vulnerable time would at the least be *four*, if the program runs for more than 10 iterations.

In Figure 5.2(c), the last section below the time-line, describes the vulnerability of each data element and the number of write-backs required in our smart cache cleaning technique. Here, we observe 67% vulnerability reduction, with only one additional write-back (compared to the EWB scheme). Here, i) data is vulnerable in the cache only for as long as it is used; and ii) every write-back operation is timed and positioned in such a way that it achieves overall vulnerability reduction. Such an adaptive scheme, that can dynamically moderate the cache-write-backs, would thus achieve power efficient vulnerability reduction on cache-data.

Quantifying the Claims

In order to quantify the claims stated above, we performed an experiment on the `matmul` program, and observed the variation in vulnerability and the number of memory writes with each of the loop orders of the program over the same set of data. From the results gathered as in Figure 5.3, we observe that the vulnerability of the data in the program cannot be studied independent of the data access pattern of the program, and therefore any method to mitigate this "data-cache vulnerability", cannot be implemented efficiently without appropriate analysis of the program's data access patterns. We propose Smart Cache Cleaning (SCC), in this work, to perform this specific data cache analysis and present an energy efficient mechanism to reduce data cache vulnerability in embedded systems.

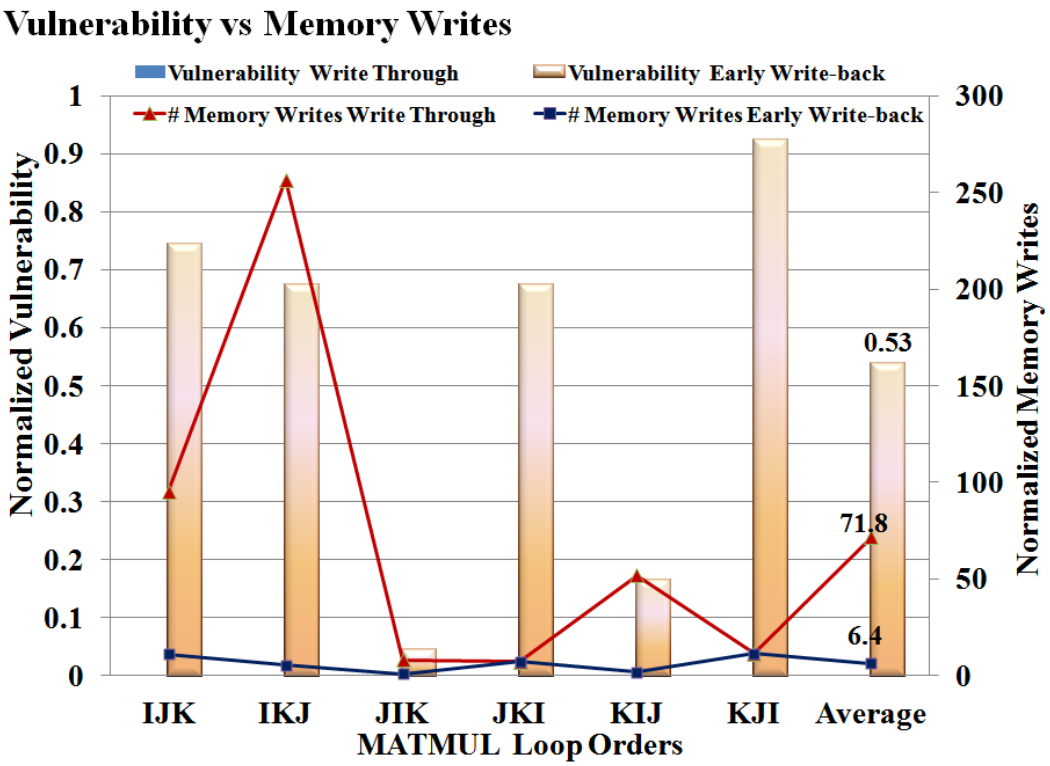


Figure 5.3: Relative comparison of vulnerability (in byte-cycles) and the number of memory-writes incurred during the implementation of write-through cache, and early-writeback on a simple matrix multiplication program. The vulnerability in the presence of early-writeback varies with the loop order, which demonstrates the room for smart cache cleaning on programs by analyzing the data access patterns.

5.4 Related Work

Careful selection and screening of materials [9], SOI fabrication technologies [16], increasing the transistor size or adding gated resistors [91] are some hardware techniques proposed to reduce soft errors in SRAM cells. In addition to the chip area and power overheads, the cost of design and fabrication of such device/circuit level techniques, and the yield obtained upon manufacture, over-weighs the reliability achieved. This reduces its applicability and/or wide-spread use of such methods to protect embedded systems. At the architecture level, ECC based techniques like SECDED [45] provide a means to protect the caches by storing ECC codes for every cache block and checking the same for correctness when used by the processor. In this, 8 *check bits* are required for every 64 bits of cache-data, which involves a 12.5% increase in the size of the cache. In addition to this, additional logic, to generate and verify the ECC codes of the read/written data, is added to the cache read-write path. Li et al. [62] in their work indicate that the hardware costs (area, performance and power) incurred in the implementation of such an ECC based error detection and correction technique is unacceptable for embedded systems. Sridharan et al [97], propose selective re-fetching of cache lines combined with a write-through cache

implementation to achieve 85% reduced cache vulnerability at the cost of 2.5% power, 7% performance and 15% chip area overheads. Authors in [63], propose to use a fixed interval early write-back technique to periodically clean the dirty cache lines and reduce their vulnerable lifetime. In spite of the reduced hardware overhead involved in its implementation, such a technique has been shown to achieve an effective trade-off between vulnerability and power only for large caches (hundreds of MB or GBs). Zhang [106] in his work, proposes two hardware based techniques (LRU and Dead-time based prediction scheme), to vary the periodic interval between write-backs from the L1 cache to the underlying memory. In this, the methodology used does not acknowledge the availability of 1-bit parity based error detection hardware in almost all modern processors, thereby underusing the available resources. In addition, for the implementation for such a smart hardware only scheme, the additional hardware required would add to the additional write-backs executed, thereby adding to the total hardware performance and energy costs to the system. In this work, we aim to achieve increased reliability in a system, by utilizing the available resources in the system, with minimum additional hardware, performance and/or energy costs. We also show through experiments over varying range of periods, that such a hardware technique when implemented in an embedded processor, has a significant impact on the number of cache write-backs and thereby adds to the power consumption of the system and also affects performance.

Software solutions are preferred as they can be implemented on existing architectures. The authors in Chapter 4 we develop Cache Vulnerability Equations (CVE) to determine statically the vulnerability of a program for a particular cache configuration. We motivate on this understanding of data reuse and cache vulnerability to develop a profile analysis techniques to determine important store references and accesses that have to be cleaned to achieve vulnerability reduction with reduced memory writes.

Software-hardware hybrid techniques have the advantage of reduced architecture overhead and the flexibility and accessibility to hardware structures aided by software techniques. Chen et al [22], propose a compiler based technique to determine the critical data used in the application and enable error correction techniques(ECC) for only those data elements. Partially Protected Caches (PPC) in which a portion of the cache is protected against soft errors can achieve around $47X$ vulnerability reduction in data intensive multimedia applications [61]. Lee et al. [59] then propose compiler techniques to statically partition data into critical and non-critical, to further enhance the protection available in a PPC. In this work, we determine the *right data* to clean and *exactly when* to do so through memory profile analysis, and with the help of hardware support, ensure that vulnerability reduction is achieved with reduced energy overhead.

5.5 Our Approach

Key Idea

Copying a dirty cache block into the memory, through write-backs (cache cleaning), reduces the vulnerability of the system but incurs an energy overhead due to memory accesses. To reduce energy overheads while also increasing reliability in a system, a prudent decision has to govern each cache cleaning operation ensuring that a memory access is performed *iff* significant vulnerability can be reduced. In embedded applications, such prudence can be achieved through profile based techniques which help in identifying the right references and the right instances that cache-data has to be cleaned.

Overview

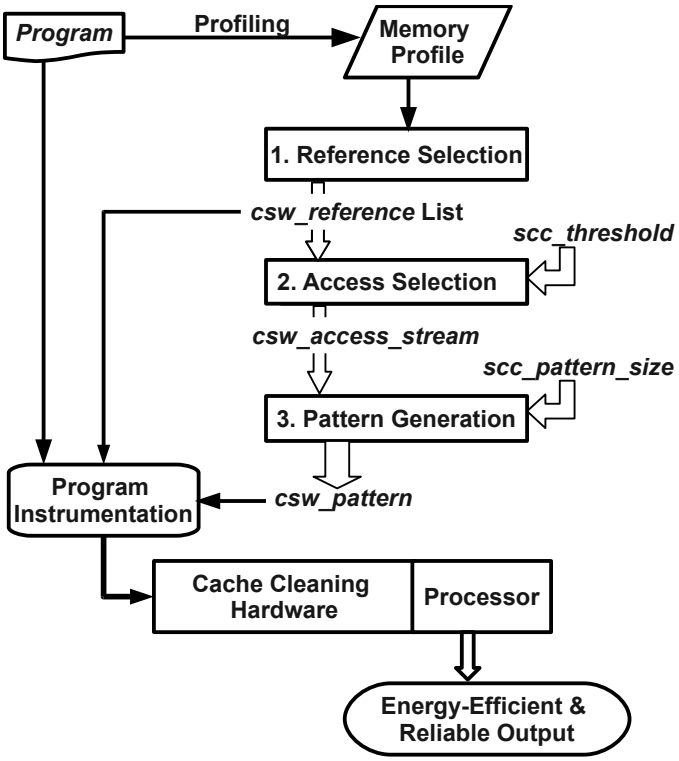


Figure 5.4: Our 4 stage Smart Cache cleaning methodology.

The memory profile information of a given program is used to evaluate the vulnerability per cache write-back profit metric for each data-reference (store instruction) in the program in the *Reference Selection* stage. For each reference in the list (*scc_reference* list) and a given threshold *scc_threshold* for the intermediate vulnerable time (IVT) generated during accesses, the list of instruction accesses to be cleaned are identified. This decision formed as a bit stream for each reference is represented by a k-bit pattern, where $k = scc\_pattern\_size$. The instruction addresses and their corresponding representative k-bit patterns (*scc_pattern*) are instrumented into the given program through compiler directives to

be loaded into their respective hardware components accordingly. With the help of hardware support from the cache cleaning architecture, the embedded processor now executes the program with minimal additional cache write-backs, and maximum reliability.

Smart Cache Cleaning Architecture

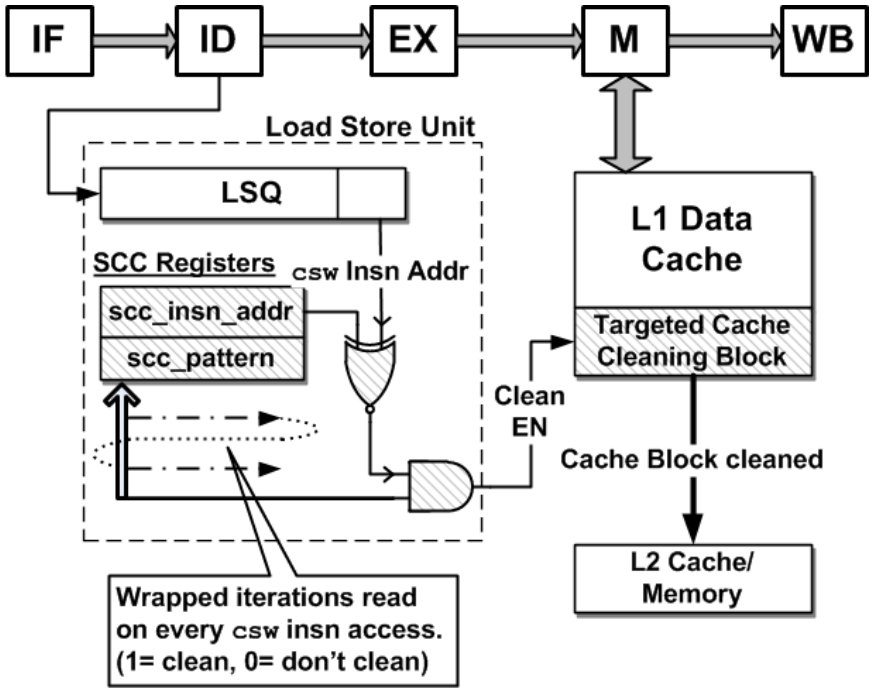


Figure 5.5: Smart Cache Cleaning Architecture: The architecture blocks as part of the SCC are shaded. Every marked store instruction (denoted by `csw`) is compared and based on the cleaning decision read by the iterator, *Clean EN* is signaled triggering targeted cache block cleaning.

The shaded blocks in Figure 5.5 represent the hardware components added to implement our smart cache cleaning technique. The "SCC Register Pair" contain the instruction address (*scc_insn_addr*)and bit pattern (*scc_clean_pattern*), for the reference set to be cleaned by profile analysis. On every access to the targeted store instruction (marked as `csw` in the instrumented code), an bit-iterator iterates over the pattern in the *scc_clean_pattern* register. A 1 read by the iterator indicates (through the *Clean EN* signal) that the cache block accessed has to be cleaned (copied to the memory by a cache write-back) while a 0 indicates otherwise. The iterations on this *scc_clean_pattern* are wrapped such that the pattern is repeatedly accessed throughout the program runtime that the corresponding instruction is accessed.

The instrumented program input to the processor, contains special instructions to load SCC-data into the "SCC Register Pair" at specific points in the program based on the memory profile analysis. The remainder of this section describes in detail the 4 step procedure that generates SCC-data for program instrumentation (as shown in Figure 5.4) and thereby trigger the cache cleaning architecture blocks to

ensure energy efficient reliability. The "Targeted Cache Cleaning Block" performs the *cache cleaning* operation as follows,:

1. the target cache-block address to be cleaned is input along with the *Clean EN* signal, from the LSQ.
2. data from the specific cache block is copied, and written-back into the underlying memory. This operation is performed after the completion of the `sw` operation, and independent of the memory access thereby causing no interference to the cache performance of the system.

Overall the SCC architecture requires $1 \times (32+32) = 64$ bit register, $1\times$ 32bit XOR gate and $1\times$ 2bit AND gate. We assume here that these SCC registers are protected against soft errors by energy efficient hardware techniques for the same. We observe that the targeted cache cleaning architecture exists in most modern embedded processors in the form of cache-flush execution units, which can be modified (if required) with minimal hardware changes. It is thus evident, that the overall area and power overheads of the additional hardware components required for our SCC implementation, are minimal and negligible.

Step 1: Smart Reference Selection

The memory profile data gathered by profiling the application is used to identify the set of references that generate significant vulnerability, and therefore have to be cleaned accordingly. The key idea behind this reference selection step is that it is possible to identify the vulnerability generated by each reference individually and thereby compare references based on the vulnerability per access metric. This metric gives an estimate of the possible vulnerability-energy trade-off that can be achieved if all the reference accesses are set to be cleaned.

Every store instruction during program execution may accesses different data elements, and each such accesses renders a cache block vulnerable. This time for which the accessed cache block remains vulnerable in the cache, is defined as the Intermediate Vulnerable Time (IVT) (defined in Section 5.2) generated by that instruction access. In order to map the vulnerability of a program to the references generating them, we calculate *ref-vulnerability* for each reference, defined as the sum of all the IVT values generated by the reference during program execution. The profit metric for each reference is thus given by $\frac{ref-vulnerability}{ref-access-count}$. From the description of our cache cleaning architecture in Figure 5.5 we observe that the there is only one SCC Register Pair and therefore only one reference can be set to be cleaned at any point in time. Among the references in the program, the chosen set of references to be cleaned are those with highest vulnerability per access values (or highest profit), with non overlapping execution time-lines.

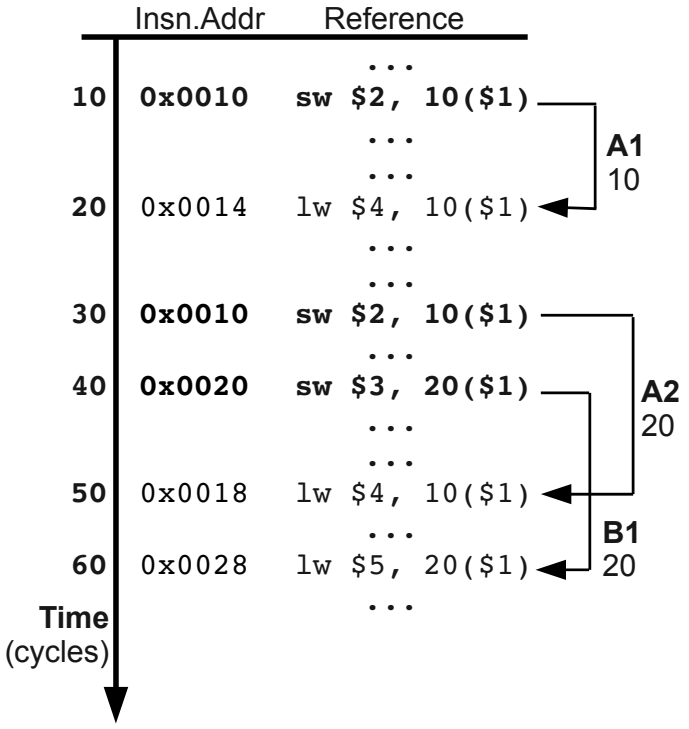


Figure 5.6: Demonstrating Smart Reference Selection: On the memory profile of a program over its execution time-line, arrays A and B are accessed by instruction addresses $0x0010$ and $0x0020$ respectively. The individual IVT values (A1, A2, B1) are labeled and annotated by arrows that connect their W and R accesses points.

The program in Figure 5.6 is of the W-R-W access pattern and the IVT values generated by reference A is $A1, A2$ and that for reference B is $B1$. From the annotated data in Figure 5.6 we derive the data table Table 5.1. It can be noted here that, the efficiency achieved due to higher profit numbers, in selecting reference B to be cleaned, automatically precludes selection of reference A under the non overlapping execution time-lines condition.

Parameters	Ref A	Ref B
ref-vulnerability	$10+20=30$	20
ref-access-count	2	1
CSW_Profit	$\frac{30}{2}=15$	$\frac{20}{1}=20$

Table 5.1: Data table derived for statistics on the example program in Figure 5.6.

Step 2: Smart Access Selection

In a program, the *ref-vulnerability* generated depends not only on its own data access pattern but also is affected by other references and data elements accessed on a set associative cache, causing cache-block replacements (cache eviction). We thus understand that not all IVT values of a reference are same owing to possible cache replacements. Having identified the most profitable reference(s) to clean, the

most profitable reference accesses can be identified as those that have IVT values greater than a given threshold value (design parameter *scc_threshold*). This threshold defines the maximum time (in cycles) that vulnerable data can remain in the cache before being evicted or over-written. For every access by the selected reference, the IVT values generated (by each access) are compared with the given *scc_threshold*, and those that exceed the threshold are slated to be cleaned. This cleaning decision is represented by a bit stream (*scc_access_stream*)of length equal to the total number of accesses by the reference, and a *Clean* operation is denoted by 1 on the bit stream and a 0 otherwise. This stream in conjunction with the *scc_reference* list, thus contains input instructions for the smart cache cleaning architecture.

Step 3: Smart Pattern Generation

Algorithm 2: SCC_BIT_PATTERN_MATCHING ()

Require: scc_access_stream ¡csw_list¿, scc_clean_pattern_size K.

1: **for** k from 0 to K **do**
2: $k\_ones \leftarrow$ Count 1^s in csw_list
3: $k\_zeros \leftarrow$ Count 0^s in csw_list
4: $Cost\_of\_0 \leftarrow k\_ones \times 1$
5: $Cost\_of\_1 \leftarrow k\_zeros \times 2$
6: **if** $Cost_of_1 \leq Cost_of_0$ **then**
7: $BitPattern[k] = 1$
8: **else**
9: $BitPattern[k] = 0$
10: **end if**
11: **end for**
12: **return** *BitPattern*

The bit stream (*scc_access_stream*) represents the set of accesses that have IVTs greater than a threshold for a reference that has been identified to have the highest vulnerability per access metric. In order to implement cache cleaning based on this bit pattern, multiple and complicated load instructions are required to ensure that the correct pattern is loaded into the *scc_clean_pattern* register for the corresponding instruction access. Therefore, a bit pattern of size k (a design pattern defined by *scc_pattern_size*), has to be determined that best represents the bit stream *scc_access_stream* of the reference accesses. In line with our intentions to ensure smart energy efficient cache cleaning, we use the *SCC_Bit_Pattern_Matching* algorithm to analyze the bit stream and derive a representative k bit pattern.

The *SCC_Bit_Pattern_Matching* algorithm described in Algorithm 2 reads the given bit stream and using a moving window of size k bits, the number of $1's$ and $0's$ in each bit position are calculated. Using these numbers, a cost is associated ($Cost\_of\_0$ or $Cost\_of\_1$) with each bit position that represents the cost of representing the bit as 1 or 0 respectively. For example, for a given bit stream, if a particular bit position in the k sized window, has many $0's$, it is right to assume that majority of these reference accesses don't generate vulnerability greater than the threshold and will therefore deliver low vulnerability savings

for the energy cost, represented by the $Cost\_of\_1$ calculated. Therefore, giving precedence to energy savings, we ensure that a bit is represented by 1 *iff* the $Cost\_of\_1$ is less than twice the $Cost\_of\_0$ value. The costs associated with the bit positions thus ensures that the resultant k bit pattern is an energy efficient representative of the given bit stream.

Step 4: Program Instrumentation and Execution

From the memory profile of a program, after the first 3 steps, a *scc_reference* list is identified, and then for each reference in this list, a representative k bit pattern *scc_pattern* is generated. The program is then instrumented with these two inputs so as to instruct the processor hardware to load corresponding values into the "SCC Register Pair". Using the memory profile, access points of the first and last accesses for each reference in the list can be identified, and at these points, corresponding load instructions are introduced with the respective reference address and k bit pattern data. It should be noted here that these instructions are compiler-directives and will not be executed through the processor pipeline, thereby involving negligible performance variation.

Cache Cleaning on Multiple References

Our smart cache cleaning architecture is scalable over the number of references set to be cleaned simultaneously. In the above discussion, we illustrate the use of only one SCC register pair (*scc_insn_addr*, *scc_clean_pattern*) while additional register pairs will enable the hardware to support multiple references to be cleaned over overlapping execution time-lines. For this purpose, the only modification in the profile analysis will be, at step 1 where references are selected such that n references may overlap in their execution time-lines, thereby allowing for corresponding access stream and k bit pattern generation. It should be noted here that additional hardware structures involve additional area and power overheads ($32 + 32 = 64$ bits for each SCC register pair added), which does not add significantly to the existing architecture.

5.6 Experiments and Results

Experimental Setup

For our experiments, we model an embedded system with a RISC processor, an on-chip L1 cache and off-chip SDRAM memory. The SimpleScalar [14] `sim-outorder` cycle-accurate simulator is configured to model the Intel XScale [48] processor architecture, with a 2-way set associative L1 cache (size = 4KB). The simulator is instrumented with code to accurately evaluate vulnerability of data used in the program (in byte cycles). To estimate memory access power, we use power numbers from the MICRON MT48V8M32LF SDRAM on an Intel 440MX chipset[94] to represent the off-chip components of the system. The energy per memory access is composed of data bus energy (9.46 nJ per burst) and SDRAM

energy (32.5 nJ per read/write burst). The power consumed during memory accesses is given by the product of total number of memory accesses and the total energy per memory access (41.96 nJ). To experimentally demonstrate the effectiveness of our SCC methodology, the SimpleScalar `sim-outorder` simulator is modified to include the Smart Cache Cleaning Architecture blocks (described in Section 5.5) and also recognize our instrumented program instructions (`csw` instructions).

To compare the trade-off between vulnerability and energy on a one dimensional scale, we use the product of vulnerability (in byte-cycles) and memory access energy (in nJ) to form Energy Vulnerability Product (EVP). Here EVP provides us with a single metric to quantitatively compare the impact of the various configurations on both vulnerability and energy consumption, thereby allowing us to achieve the required balanced trade-off. In any application, the data acceses on arrays within nested loops, are the program segments that contribute to data-cache behavior. We perform our experiments on benchmark loops from *LAPACK*[4] and *LiverMore Loops*[70], which are scientific, data-intensive and computation-intensive benchmarks representative of applications executed on such embedded systems. To compile our benchmarks we used GCC (v 2.7.3) with all optimizations turned on. In our attempt to analyze the efficiency and impact of SCC, we experiment over each benchmark varying all the possible design parameters like *scc_threshold* ($5, 10, 15, \cdots, 200$ cycles), *scc_pattern_size* ($4, 8, 16, 32$ bits) and the number of *scc_insn_reg* registers (number of references to clean). We then compare the EVP values thus obtained with that of a write-through cache and early-writeback (EWB) cache configuration of varying periods ($100, 200, \cdots,$ 2000 cycles).

Better Energy-Vulnerability Efficiency With Smart Cache Cleaning

The graph in Figure 5.7 plots EVP values, normalized to that of the original program, obtained for the best EWB configuration and best SCC threshold values. For each benchmark, among the results obtained for the set of EWB periods experimented ($100, 200, \cdots, 2000$ cycles), we choose the one with the least EVP value. Similarly, from the results for varying *scc_threshold* values, we choose the threshold that delivers lowest EVP. The graph clearly demonstrates that overall the benchmarks, the SCC technique achieves lower EVP and therefore better energy efficient vulnerability reduction. From the graph in Figure 5.7 we observe:

1. The second bar (labeled "SCC (1 Reg)") represents normalized EVP values of the SCC technique obtained for experiments using 1 *scc_insn_reg* register, showing the efficiency obtained when only one reference is selected (to be cleaned) at any point in time. For most benchmarks this bar remains unseen owing to their significantly low EVP values ($\leq\ 2 \times 10^{-7}$) indicating highest possible

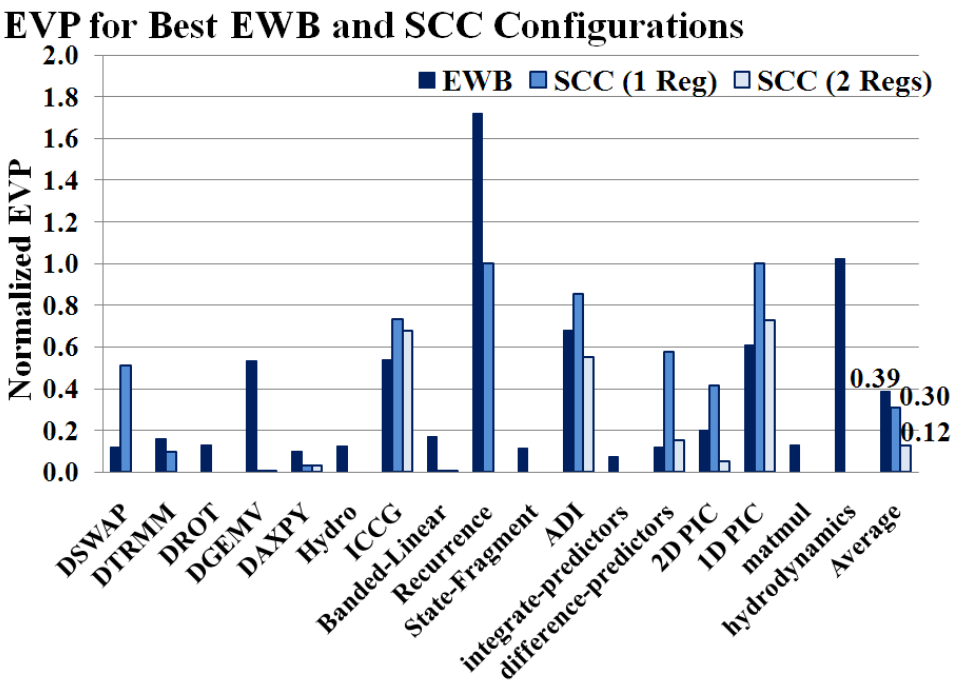


Figure 5.7: The graph showing Normalized EVP of the best EWB period (EWB configuration with least EVP) and the best *scc_threshold* parameter using 1 and 2 *scc_insn_reg* registers, demonstrates higher energy-vulnerability efficiency with the SCC technique.

efficiency.

2. In the case of benchmarks like *ICCG, ADI, 2D PIC, 1D PIC* and *diff-predictor* the program contains multiple references executing in overlapping time-lines with comparable vulnerability per access profits, and therefore the selection of only one reference seems insufficient. For majority of the benchmarks experimented we observe that the use of a second *scc_insn_reg* register decreases the EVP significantly, which is represented by the third bar (labeled "SCC (2 Reg)").

3. From the results for the *Recurrence* benchmark, we see that the early writeback mode of cache cleaning loses on EVP by 60% compared to the original program. Owing to the complex data access pattern in the program, the best early write-back configuration (EWB period = 1800 cycles) achieves only 26% vulnerability reduction at the cost of $2X$ increased memory writes. On the other hand, having the knowledge of the data access pattern and the flexibility to enable cache cleaning only at instances that achieve profitable vulnerability reduction, the SCC technique using one register achieves 26% vulnerability reduction at $< 1\%$ increase in memory writes. Moreover, with the use of an additional register, we achieve 100% vulnerability reduction at 96% increase in memory writes.

4. The average EVP plots, towards the right end of the graph indicate that our SCC technique using one *scc_insn_reg* register is 8% lesser, and using two registers is 26% lesser than that of the EWB technique.

More Energy Efficient Design Points in SCC

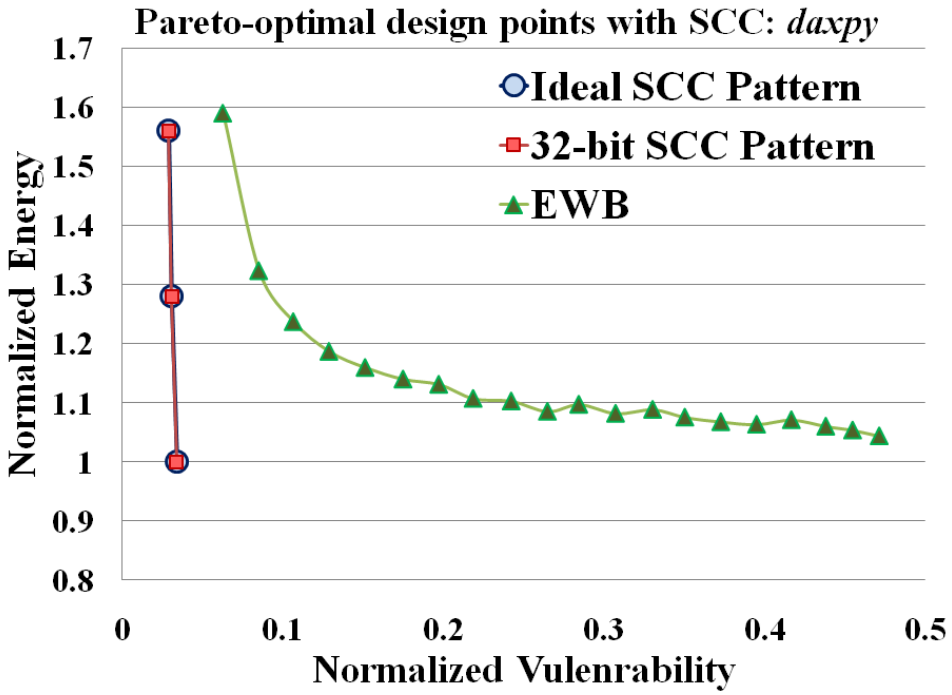


Figure 5.8: Normalized vulnerability and energy plots for two benchmarks across varying *SCC_Threshold* values. The plots for *32-bit Pattern SCC* closely follow that by *Ideal SCC* in *DSWAP*, while they overlap in DAXPY, demonstrating the accuracy of Algorithm 2. Vulnerability and energy trade-off is observed for varying threshold values for each benchmark.

For each benchmark, we experiment over varying thresholds and for each perform experiments using the ideal *SCC_Access_Stream* derived after memory analysis and again using the K-Bit pattern that best matches with the access stream (determined using the pattern matching algorithm Algorithm 2). In Figure 5.8, The x-axis plots the normalized vulnerability values, while the y-axis plots normalized energy (number of cache write-backs). For one benchmark, in the EWB configuration, the early write-back period is varied $(100, 200, \cdots, 2000)$ and the normalized vulnerability and energy overhead incurred are plotted in Figure 5.8 labeled EWB. Similarly, for the same benchmark the vulnerability and overhead values are plotted for varying threshold values $(5, 10, 15, \cdots, 200)$, using our SCC technique. In the graph plotted in Figure 5.8, each plotted point is a design point in a design space exploration to determine the right EWB period or *SCC_Threshold* to choose for energy efficient vulnerability reduction. A design point closer to the x-axis denotes that it has a low energy overhead, and a point closer to the y-axis denotes low vulnerability (or increased reliability) of the program. A point that is close to the origin

(0,0) is the most efficient point which denotes least vulnerability at least energy overhead. It can be clearly seen here, that the results from our SCC technique over varying threshold parameters have points more closer to the x-axis and also more closer to the y-axis than any other point in the EWB plot; thereby demonstrating the energy efficiency realized. In other words, we say that design points obtained by our SCC technique are pareto-optimal to design points achieved by hardware techniques like WT or EWB.

Generated K-bit pattern achieves close-to-ideal SCC efficiency

The algorithm *Generate_Bit_Pattern* defined in Algorithm 2, uses a weighted matching technique to analyze the ideal access stream (*SCC_Access_Stream*) of a reference selected to be cleaned, and represents the same as a k-bit pattern. In our experiments we evaluate the accuracy of the algorithm over *SCC_Pattern_Sizes* $4, 8, 16$ and 32. In Figure 5.8, the normalized vulnerability and energy values for DAXPY, across varying threshold values, are plotted for a pattern size of 32bits. It is evident from the overlapping plots of SCC values in Figure 5.8, the values obtained after pattern matching on a 32-bit register closely follow that of the ideal access stream (*CSW Access Stream*). This demonstrates the accuracy of our Smart Pattern Matching algorithm (Algorithm 2). We again observe that for larger pattern sizes, the extent of matching accuracy increases, but when implemented does not show any significant difference in the vulnerability and energy numbers. The system designer is thus able to choose between allowing one 32bit register or 2 16bit registers based on hardware constraints, and still achieve intended vulnerability reduction.

EVP decreases with increase in references to clean

When larger number of SCC registers are integrated into the system, our SCC technique provides for scalability and therefore energy efficient vulnerability reduction in the system. Figure 5.9 plots the EVP values of four benchmarks for varying numbers of references selected to be cleaned simultaneously. For each benchmark, the maximum number of references allowed is determined through memory analysis. We observe here that, in each benchmark the small additional SCC registers, translate into significant EVP reduction. It is interesting to note that in the *Diff-Predictors* benchmark, with the choice of $2, 3$ and 4 registers the greedy nature of selecting references to clean translates into greater EVP numbers, however as the number of selected references increases to 7, the EVP is significantly reduced.

Increase in the number of references selected to be cleaned simultaneously maps to an increase in the number of *scc_insn_reg* and *scc_pattern* registers in the cache cleaning hardware. In the case that greater number of such registers are allowed in the system, our SCC technique provides for scalability and thereby more energy efficient vulnerability reduction in the system. Figure 5.9 plots the EVP values of four benchmarks for varying numbers of references selected to be cleaned simultaneously. For each

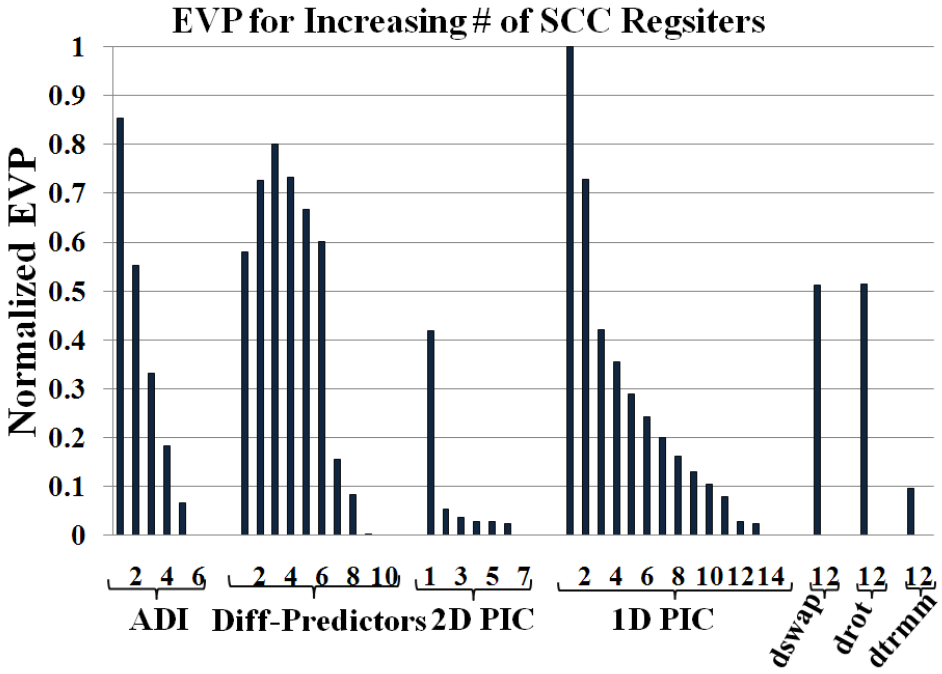


Figure 5.9: The EVP of the application reduces with increase in the number of *scc_insn_reg* registers used.

benchmark, the maximum number of references is given by that determined through memory analysis. We observe here that, in each benchmark the small additional hardware registers translate into significant EVP reduction and thereby efficient reliability of the system. It is interesting to note here that in the case of *Diff-Predictors* benchmark, with the choice of 2 and 3 registers the greedy nature of selecting references to clean simultaneously translates into bad EVP results, but when the number increases to 7, the EVP reduction is significant. The metric used to sort each reference is nothing but the vulnerability per cache-writeback ratio, which from this example shows a degree of inaccuracy. A more intelligent metric and selection process would guarantee efficient choices in parameter values. We intend to pursue this direction in our future work.

5.7 Software-only Smart Cache Cleaning

As described in Figure 5.5, additional hardware is required for the implementation of the proposed Smart Cache Cleaning, as a hybrid software-hardware profile based technique. Though the added hardware is relatively insignificant, the accesses of these hardware in conjunction with each cache access indicates additional power overheads. In order to eliminate the need for the SCC hardware to load the bit pattern, and then iterate over each bit for each cache access by the specified `store` instruction, in this work, we propose a pure software-only solution to achieve the same effects. The given loop can be unrolled and thus we have *k* instances of the specified `store` instruction as chosen from the *SCC_Analysis*. Among the

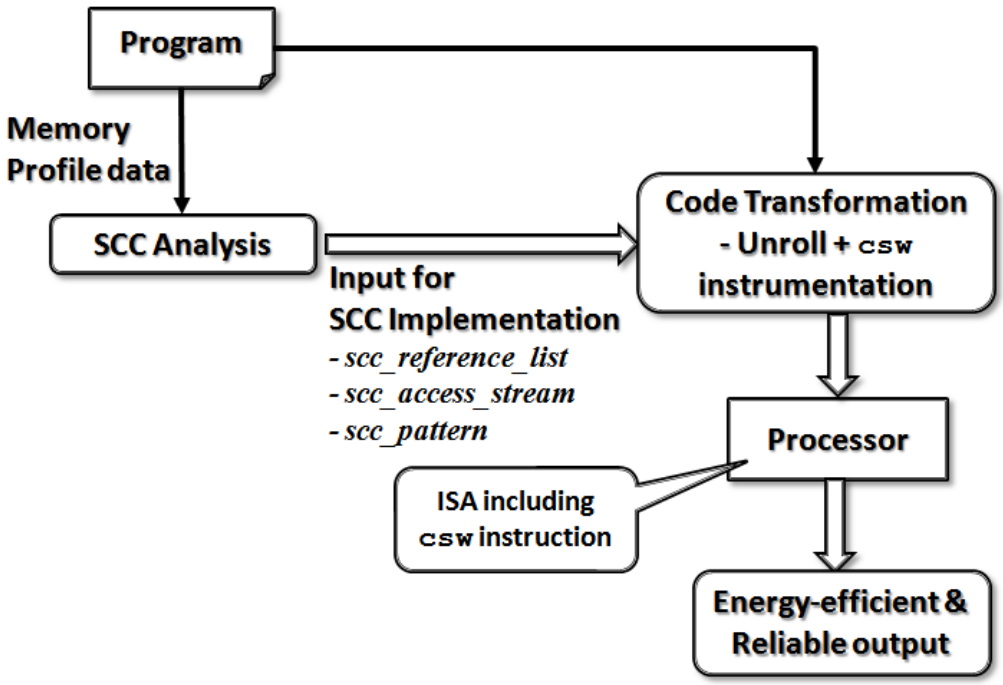


Figure 5.10: Software-only SCC Overview: Overview of the software-only SCC technique is described.

k instances, the derived *scc_pattern* determines the instances that have to be cleaned, and those that are not to be cleaned. The instances that need to be cleaned can be replaced by a new instruction `csw` - clean and store word. In this, no significant decode or control logic needs to be added for its implementation. In addition to performing the `store` operation, the cache-flush logic within the processor is triggered for the targeted cache block. This performs the required cache cleaning on the specified instance on the specified cache block. Figure 5.7 describes this software-only solution as a block diagram. Figure 5.7 plots the experimental results as a proof of concept that our software-only solution achieves the same benefits as that of the hybrid SCC technique. In this case, we also see instances where the performance of the loop can be improved owing to the instruction scheduling on a superscalar processor made possible by the loop unrolling code transformation.

5.8 Summary

The cache, occupying the majority of chip area and with its unique architecture and circuitry is the most susceptible architecture component in the processor. By reducing the time that vulnerable data resides in the cache, we can reduce the probability of an error in cache data and thereby reduce the overall system failure rate (or improve system reliability). Hardware based mechanisms like write-through cache or early write-back cache though efficient in reducing the vulnerable data time in the cache, incurs a large energy overhead due to increased L1-memory writes. we develop a hybrid hardware-software Smart

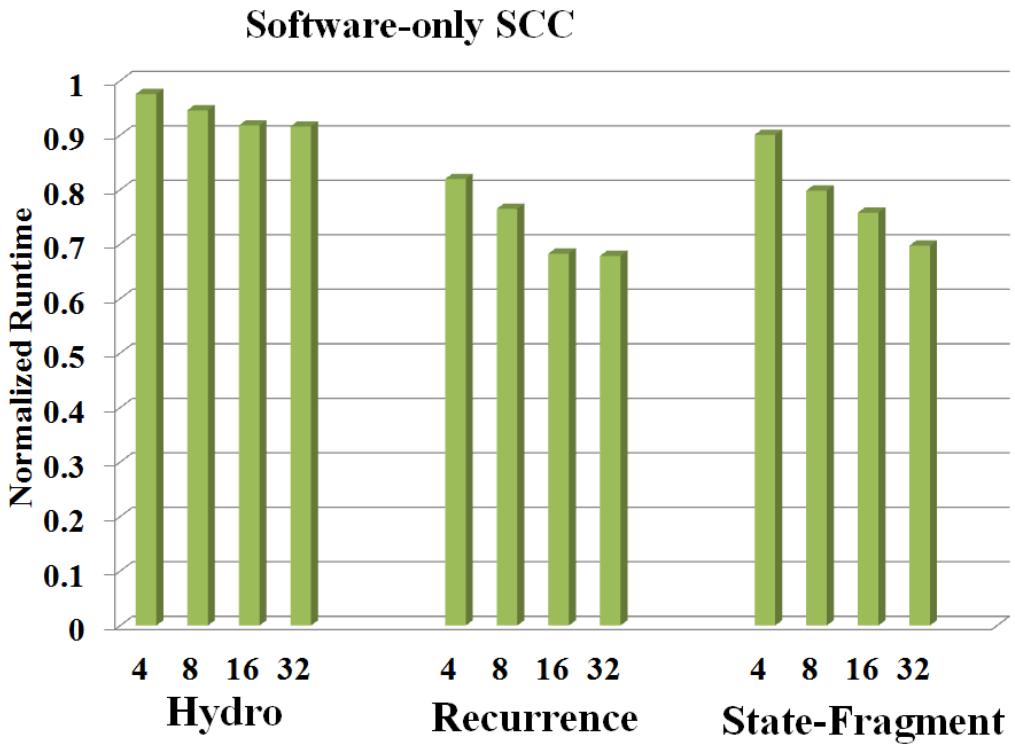


Figure 5.11: Experimental Demonstration: The EVP of the application using SCC implementation on varying orders of unrolled loops, which correspond to the different k-bit pattern sizes identified through the *SCC_Analysis* performed.

Cache Cleaning (SCC) technique, where we use the memory profile of an application to accurately estimate data vulnerability (time that updated data is in the cache), identify the program instances that generate the same. We then enable cache cleaning on specific cache blocks at specific instances, to ensure energy efficient reduction of data cache vulnerability. Our experiments over scientific benchmarks show that when compared to the hardware based early write-back cache architecture, the SCC technique achieves 26% lower Energy Vulnerability Product.

Our profile base method currently identifies references with higher profit on non-overlapping execution time-lines, but it is observed that for some applications, the references are accessed in bursts. In such a case, the use of a reference to clean can be interleaved with another to achieve better results. It is possible to analyze loops with affine access functions statically at the compiler for its vulnerability and thereby identify the right references and access instances to perform cache cleaning. Intelligent schemes can be devised to analyze the data access patterns statically, and thereby derive the design points for varying *threshold* and *k-bit* values. Such a methodology will help develop a well-rounded and automated scheme to implement smart cache cleaning.

Chapter 6

HYBRID COMPILER MICRO-ARCHITECTURE TECHNIQUES: FOR POWER-EFFICIENCY

"In an attempt to reduce system power consumption, researchers have developed a wide array of techniques at the hardware micro-architecture level [37, 83, 107], and compiler levels [27, 80] of embedded design abstraction. By infusing the knowledge from one design layer over another, the combined advantage has been demonstrated to improve system benefits achieved. Such compiler micro-architecture hybrid techniques developed have been proved to facilitate improved benefits to the system in the form of relatively reduced hardware overhead, and increased energy reduction through the use of smart compiler optimizations."

In this chapter, we demonstrate such a hybrid technique to reduce the energy consumption of the Translation Look-aside Buffer. In this, we develop code transformation techniques to reduce the page switchings in data cache accesses and propose an efficient page-aware code placement technique to enhance the energy reduction capabilities achieved by the *Use-Last* TLB architecture for instruction cache accesses.

6.1 Introduction

The Translation Look-aside Buffer or TLB is an important component of high-end multi-tasking embedded processors, like the Intel XScale [48]. The TLB performs virtual to physical address translation and determines page access permissions. Most modern processors, including the Intel XScale implement virtually-addressed caches, in which the cache lookup is directly performed on the virtual address provided by the processor, and therefore the TLB lookup comes in the critical path. Elkman et al. [30] note that the TLBs can consume $20-25\%$ of the total $L1$ cache energy. Kadayif et al. [52] observed high power densities of the data-TLB, as compared to the data-$L1$ cache. Thus reducing the power consumption of TLBs is an important research problem. In [52], researchers show that the iTLB architecture has a power density of $7.820\ nW/mm^2$ compared to 0.975 and $0.670\ nW/mm^2$ for $iL1$ and $dL1$, respectively.

Several TLB designs have been proposed to trade-off the TLB lookup delay, area and power consumption [83, 107]. One simple, yet effective technique for TLB power reduction proposed in [26, 37], is the *Use-Last* TLB architecture. Observing that there is a high probability that instruction access will refer to the same page as the last one, they store the previous page translation information into a latch, and thereby reduce the TLB lookup power. The Use-Last TLB architecture is able to reduce the instruction TLB power by 75%. However, since data accesses do not exhibit as high locality as instructions, this micro-architectural technique was not effective for data TLBs.

6.2 Data-TLB Power Reduction

Observing that there is a high probability that instruction access will refer to the same page as the last one, they store the previous page translation information into a latch, and thereby reduce the TLB lookup power. The Use-Last TLB architecture is able to reduce the instruction TLB power by 75%. However, since data accesses do not exhibit as high locality as instructions, this micro-architectural technique was not effective for data TLBs.

We develop compiler techniques to reduce the power consumption of the Use-Last TLB architecture by improving the locality of data accesses. We propose a novel instruction scheduling and operand reordering technique, heuristic for deciding when to perform array interleaving, and loop unrolling to minimize the page switchings between consecutive TLB accesses while minimizing performance loss. Our combined technique can reduce the TLB switches by 39%, with minimal performance impact on benchmarks from MiBench, Multimedia, DSPStone and BDTI suites. Note that this improvement is above and beyond what the Use-Last hardware technique alone could achieve.

Related Work

Several researchers have proposed efficient circuit-level, micro-architectural and software techniques to reduce the power consumption of the TLB and the Memory Management Unit.

Compiler based Approaches

A compiler-directed array interleaving technique [27] was proposed to save energy in multi-bank memory architectures with power control features. In this, the arrays used in separate banks are interleaved such that only one of the banks is active and the other can be powered down, thus saving energy. The energy reduction achieved by this technique does not account for the leakage power of the SRAM cells during standby mode. Parikh et al in [80] schedule instructions within a block based on the minimum obtainable value for a weighted cost function:*circuit-state cost*. One recent work is [24], where energy reduction is achieved through effective utilization of resources by switching between two processor modes based on the cache misses.

Closest Approach

The work closest to our approach, is by Kandemir et al. [53]. Their compiler technique is to increase the effectiveness of a previously proposed architectural technique that uses *Translation Registers* or TRs. The addition of TRs requires changing the ISA, which may not be desirable in many cases. In contrast, our approach is to improve the effectiveness of Use-Last TLB architecture, which exists in the Intel XScale processor. They have to profile the code to find out which page will be accessed frequently in

the near future, and then generate code to load the translations to that page into TRs. In comparison, our approach is a static technique. We do not need/use profile information. Not only that profile-based compilation is limited in application and scope, it has huge overhead in terms of compilation time. Our technique does not have any such overheads. Finally, in their technique, the code is modeled as nodes which represent loop nests that access data from a particular page region. Code transformations to enhance the use of TRs are directed at scheduling these loop nests (nodes that access data from a particular page region) together. In contrast, our approach is to schedule and transform instructions so that the accesses to the same page are grouped together. Our technique operates at a finer granularity than theirs, and could therefore co-exist, and enhance the effectiveness of each other.

Use-Last *TLB Architecture*

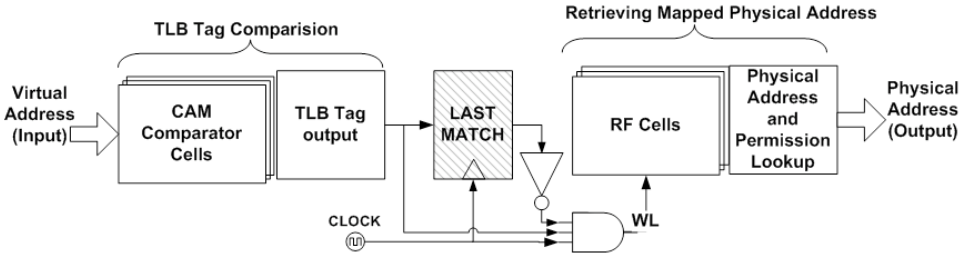


Figure 6.1: Representative block diagram of the *Use-Last* TLB Architecture [37]

Our compilation approach enhances the effectiveness of an already effective and popular TLB architecture, the *Use-Last* TLB architecture. Proposed in [37], the Use-Last TLB architecture utilizes a modified TLB-CAM structure (Figure 6.1. The virtual address input is matched with the TLB tag through the CAM cells (which has reduced power consumption). The TLB tag is then used to retrieve the mapped physical address from the register files. The lookup on the register files is a power consuming process because of the bit-line and word-line drivers and other associated circuitry involved in its operation. The key factor in this architecture design is the latch used to store the tag address of the previously accessed address. If the two TLB tag addresses match, the page address and access information at the output is the same for both. In this case, the word line (WL) of the register files are not activated and the switching energy of the RF cells and associated circuitry is eliminated. The effectiveness of this technique was demonstrated on instruction TLB, and it was shown to reduce the power consumption by 75%. However, this technique was deemed un-useful for data caches, as data accesses in general do not exhibit high data locality as compared to instruction TLB. Our work aims to enhance the effectiveness of this architectural technique on data caches through code transformations and achieve power savings through reduction in the number of page-switches during successive data accesses.

Experimental Setup

We explore and develop compiler techniques for the Intel XScale processor [48] on which the *Use-Last* architecture was implemented (as described in Section 6.2). Intel XScale is an out-of-order, 7-stage superpipelined high-end embedded processor, which runs at up to 1 GHz. The Intel XScale uses TLBs to implement virtual memory support. The Intel XScale is intended to be used in wireless and handheld applications and therefore we execute benchmarks from MiBench [36], MultiMedia [7], DSPstone [108], Spec2000 [41], and the BDTI [11] benchmark suites. The *sim-outorder* cycle-accurate simulator of the SimpleScalar Toolset [14] was modified to model the Intel XScale memory configuration and to determine the total number of page switches in the data TLB in a program.

Page Switch-Aware Instruction Scheduling

Instruction scheduling can aggregate instructions that access the same pages consecutively, thereby reducing page switches in the data TLB. In addition, for commutative operations, it is also possible to reorder the operands, and effect the memory access pattern. We develop a combined instruction scheduling and operand reordering technique.

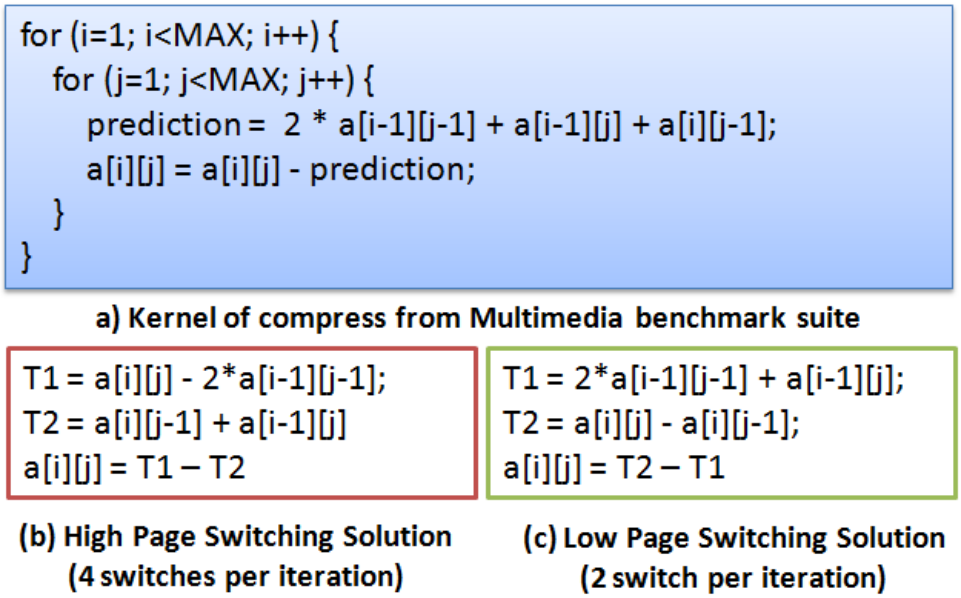


Figure 6.2: Impact of code generation on TLB page switching

Motivation

We motivate the applicability and effectiveness of fine-grain instruction and operand reordering on TLB page switches using a kernel from the *compress* benchmark, shown in Figure 1(a). The kernel accesses

elements from a two-dimensional array. If the array size is much larger than the page size(which is typically small in embedded systems), elements from the higher dimensions may reside in different pages. In this example, there are high chances that $a[i]$, and $a[j]$ may be in different pages, if $i \neq j$. Assuming this, the two code sequences generated by the compiler, illustrated in Figure 1(b) and (c), may result in the same performance, they may differ significantly in the number of TLB switches they cause. When executed, the code in Figure 1(b) will result in accesses in the sequence: $a[i][j]$, $a[i-1][j-1]$, $a[i][j-1]$, $a[i-1][j]$, and $a[i][j]$, which will result in 4 page switches per iteration, while the code in Figure 1(c) will result in only 1 page switch per iteration. Note that depending on the cache size and page size, the page switches can vary, but if there is no performance impact, it will be better to generate the code as in Figure 1(c). In the rest of this section, we first formulate the problem of minimizing the page switches by instruction scheduling and operand reordering. Finding the problem to be NP-complete, we propose a heuristic for the same.

Problem Formulation

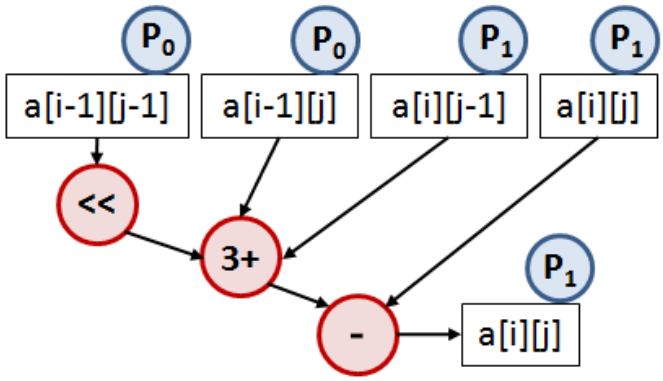


Figure 6.3: DFG and page mapping of compress kernel

Input: Data Flow Graph (DFG) is a directed acyclic graph (DAG) $D = (V,E)$ of a code sequence. The nodes $v \in V$ represent instructions $i \in I$. An instruction i is represented by a ordered $(k+2)$-tuple $i =< op,d,s_1,s_2,...s_k >$, where op is the opcode, d is the destination, and there are k source operands, $s_1,...s_k$, $d,s_1,s_2,...s_k \in O$, where O is the set of program variables, or operands. There is a directed edge $e = (v_1,v_2) \in E, \ni v_1,v_2 \in V$, from v_1 to v_2 if the destination of the instruction represented by node v_2, is the same as any of the source operands of the instruction represented by node v_1. i.e., $(v_1.i.d = v_2.i.s_1) \vee (v_1.i.d = v_2.i.s_2) \vee ... \vee (v_1.i.d = v_2.i.s_k)$. The data flow graph will also have nodes at the beginning of the graph, representing loading of operands, and nodes at the end of the graph, representing storing of operands, or intermediate values that will be carried over to the next loop. The DFG of the compress kernel is illustrated in Figure 2.

Output: Instruction Sequence represented by the function $Time : I \rightarrow \mathbb{N}$ such that all data dependencies are maintained. i.e., if there is an edge from instruction i_a to i_b, then $Time(i_a) < Time(i_b)$.

Objective: Minimize Page Switches in the instruction sequence. To estimate page switching at the compiler level, we define a function $Page : O \rightarrow P$, which maps operands $o \in O$ to pages $p \in P$, where P is the set of all the pages accessed by the application. A source operand may be a scalar, or an array, and can be defined in a local scope or a global scope. We define $Page(s)$ thus:

- $Page(s) = undefined$ if the operand s is a local scalar variable. This is because most probably all the local scalar variables will be allocated to registers and therefore will not involve in memory access.
- $Page(s) = p_0$ if s is a global scalar variable. We assume that all the global scalars are allocated to a single page.
- For the global or local arrays, we assume that each array, irrespective of it's size is mapped to exactly one unique page.

Page Switch Model In addition, we also need a page switch model, i.e., given a sequence of instructions, how many page switches will occur. We assume that when an instruction i executes, its operands are accessed in the order $\{i.s_1, i.s_2, ..., i.s_k, i.d\}$. Assuming that the page accessed just before the execution of an instruction i is p, then, we define the page switching function, $PS_I(p, i_1, ...i_n)$ to be the number of page switches when a sequence of instructions $i_1, ...i_n$ is executed.

$$\begin{aligned} PS_I(p, i_1, ...i_n) &= PS_O(p, i_1.s_1, i_1.s_2, ..., i_1.s_k, i_1.d, \\ &= i_2.s_1, i_2.s_2, ..., i_2.s_k, i_2.d, \\ &= ..., \\ &= i_n.s_1, i_n.s_2, ..., i_n.s_k, i_n.d) \end{aligned}$$

The total page switch count between operands can be recursively computed using the following equation:

$$\begin{aligned} PS_O(p, o_1, ..., o_m) &= PS_O(p, o_1) \\ &\quad + PS_O(LP_O(p, o_1), o_2, ..., o_m) \end{aligned}$$

where $PS_O(p, o) = 1$, when both p and $Page(o)$ are defined, and $p \neq Page(o)$. $LP_O(p, o)$ is the last page accessed when operand o_1 is accessed after accessing page p. The last page function $LP(p, o) = Page(o)$, if $Page(o)$ is defined, otherwise, it is p.

Solution for Page Switch Minimization

To minimize page switches by instruction scheduling and operand reordering, we define a Page Switching Graph $PSG\_full = (I, S)$, which is a directed graph, whose vertices are instructions $i \in I$, and there is an edge from instruction i to instruction j if instruction j can be scheduled immediately after instruction i. We attach a weight attribute to each edge $w(i, j)$, which is the minimum increase in the page switches when instruction j is scheduled immediately after instruction i. Thus,

$$w(i,j) = \begin{cases} min \begin{cases} PS_O(p, j.s1, j.s2, j.d) \\ PS_O(p, j.s2, j.s1, j.d) \end{cases} & \text{if } j.op \text{ is comm} \\ PS_O(p, j.s1, j.s2, j.d) & \text{otherwise} \end{cases}$$

where p is the last page that has been accessed after instruction i is executed. We add a dummy source node, and a sink node so that there is an edge from the source node to all the instructions that do not have any predecessors in DDG, and there are edges all nodes that do not have successors in DDG to the sink node. Dummy nodes access only $undefined$ pages.

The problem of finding the instruction sequence and operand ordering that minimizes the number of page switches is exactly equal to the problem of finding the shortest hamiltonian path from source node to sink node. This implies that if we can solve the problem of page switch minimization in polynomial time, we can also solve the hamiltonian problem, which is a well known NP-Complete problem in polynomial time. This is quite unlikely, therefore the problem of scheduling for page switch minimization is NP complete. Therefore we focus our efforts on developing scheduling heuristics for page switch minimization.

Heuristic for Page Switch Minimization

For heuristics, we first construct a Page-Not-Switching Graph $PNSG = (I, D, S)$, where the nodes (I) are instructions, and there are two kinds of edges, first is the set of data dependence edges D, and the second S is the set of inter-instruction page not-switching edges. Thus there is a an edge $s = (i, j) \in S$ between two instructions: $i, j \in I$, if there is NO inter-instruction page switch when instruction j is scheduled immediately after instruction i. In other words, $(i, j) \in S, \forall i, j \in I$, iff $Q_{ps} \geq 1$, where

$$Q_{ps} = \begin{cases} min \begin{cases} PS_O(p, undefined, i.d, j.s1) \\ PS_O(p, undefined, i.d, j.s2) \end{cases} & \text{,if } j.op \text{ is comm} \\ PS_O(undefined, i.d, j.s1) & \text{otherwise} \end{cases}$$

An example of a *PNSG* is shown in Figure 3. The nodes 1 through 7 are instructions, and the solid edges represent data dependencies. The dashed edges represent the inter-instruction page not-switching edges. We now perform our scheduling on this graph representation.

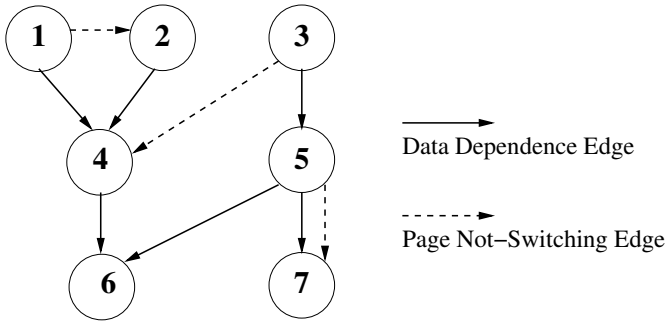


Figure 6.4: Problem in greedy solution.

We first developed a greedy algorithm. In the greedy algorithm, in every iteration, the *last scheduled instruction*, l is maintained, and list of instructions that are now ready to be scheduled, R is created. If there is a page-not-switching edge between l and any instruction $r \in R$, then r gets priority, as it minimizes the page switches. Thus suppose instructions $1,2$ and 3 are scheduled, with $l = 3$. Then R can be computed as $R = \{4,5\}$. Out of these, the greedy heuristic will pick up instruction 4.

Figure 6.4 illustrates one problem with this simple approach. In the first iteration, the greedy solution can pick up either instruction 1, or instruction 3. Picking up instruction 3 is a bad choice, because it is not possible to schedule instruction 4 as the second instruction. Instruction 3 should only be scheduled only if instruction 4 can be scheduled next. We fix this problem by adding that - when picking an instruction which is the source of a page-not-switching edge, we pick up a pair of instructions to schedule; plus, we give priority to pick up instructions that are not connected through page-not-switching edges. This gives us more opportunities to pick up instruction pairs with page-not-switching edges.

Experiments

We have implemented this page-aware instruction rescheduling algorithm as a compiler post-pass [93]. We compile our benchmarks with GCC -O3 optimization, to ensure that the benchmarks are compiled

and scheduled for the maximum performance. We disassemble the generated object file, discover the basic blocks, and re-create the control flow graph (CFG), and the data flow graph. We perform this modified list scheduling heuristic on basic blocks. This fine grain instruction scheduling approach is applicable to any program. The effectiveness of this approach could be increased by performing our scheduling on hyperblocks, and/or superblocks. We observed that our scheduling gains from performing local reordering of load instructions. There is not much increased opportunity to move load instructions across basic blocks, because of tight data dependencies.

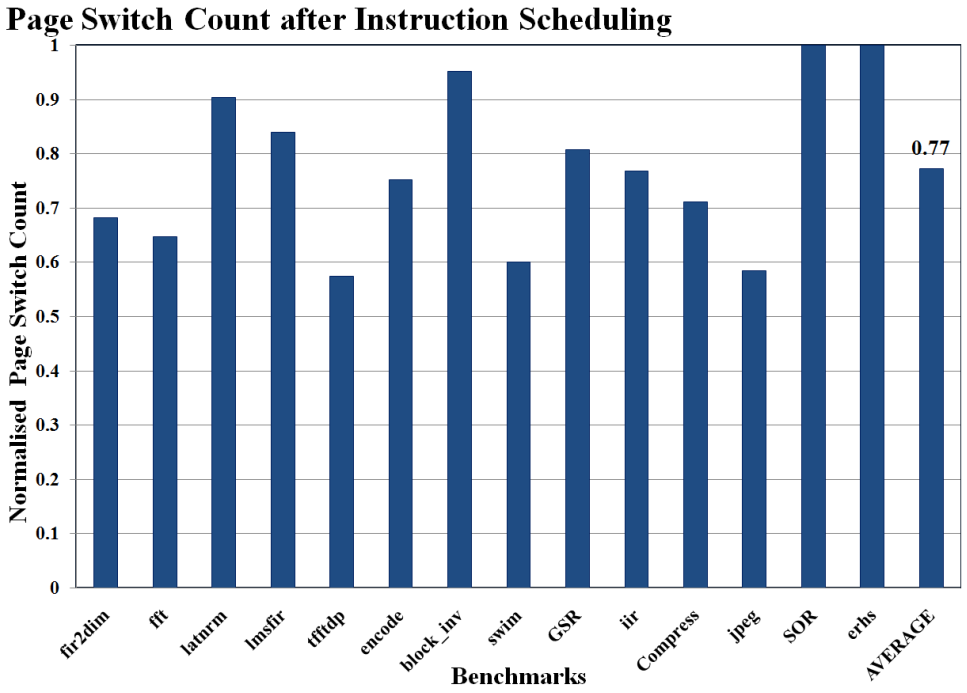


Figure 6.5: Impact of Instruction Scheduling on Page Switch Count

We modified the sim-outorder [14] simulator to count the page switches for an application execution. Figure 6.5 plots the page switch count, after implementing our page-aware instruction scheduling and operand re-ordering transformations normalized to the baseline page switch count. On an average, our technique achieves 23% reduction in the page switch count as indicated by the right-most bar in Figure 4. As a matter of fact, we observed an average performance improvement of 4%. This reduction in page switches directly translate into 23% power savings in the Use-Last TLB. Note that this is over and above what Use-Last TLB architecture achieves on its own.

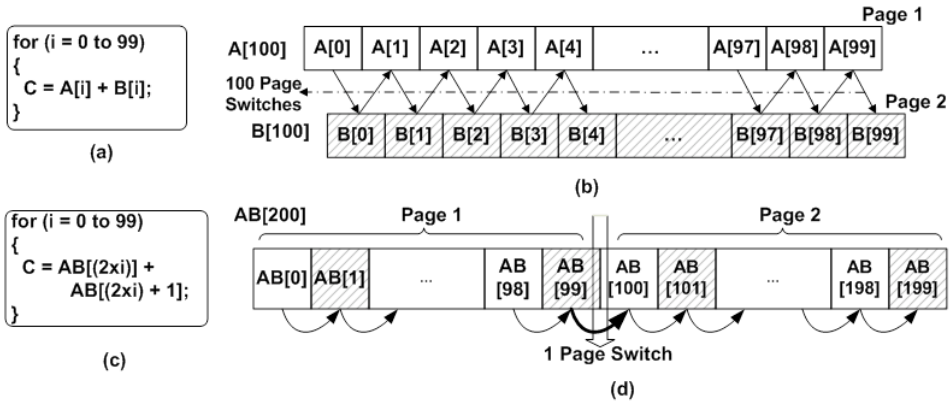


Figure 6.6: Array Interleaving through example: (a)Example loop;(b)Array allocation and access pattern; (c)Loop block with interleaved arrays; (d)Array allocation and access pattern of interleaved array

Page-Switch Aware Array Interleaving

Motivational Example

Figure 6.6 shows how array interleaving can reduce the TLB page switching over data accesses in the program. The code in Figure 6.6(a) shows a loop which accesses elements from two different arrays A and B, which are mapped to different pages. Figure 6.6 (b), shows that when this loop executes, there is a page switch between consecutive memory accesses in the program. Figure 6.6(c) shows the transformed code after interleaving. Array interleaving places the elements of the two arrays as alternate elements of the array AB. Figure 6.6(d) shows that there is no page-switching between consecutive access to AB.

Identify the Arrays to Interleave

The problem of reducing TLB page switching is localized to consecutive memory accesses, therefore interleaving of arrays need only be directed to decrease the page switching in the innermost loop. Consider a nested loop of 3 levels, whose iterators are i, j, and k, in which are there are references to arrays A and B. Suppose in the innermost loop, the reference functions are affine functions of the iterators, i.e., the access function can be represented as a linear combination of the iterators, $f_A = a_0 + a_1 i + a_2 j + a_3 k$, and similarly $f_B = b_0 + b_1 i + b_2 j + b_3 k$.

We consider two arrays A and B as interleaving candidates only if i)the access functions of the arrays are the same. Thus, $a_0 = b_0, a_1 = b_1, a_2 = b_2, a_3 = b_3$ ensuring minimized page switches after interleaving. ii)the arrays of the same size. For example, we will interleave an array of integers with another same size array of integers. It is important to note that while it is possible to interleave arrays with slightly different access patterns also, it results in an overhead in terms of extra addressing instructions. However, the innermost loop may contain several references to the same array. Two arrays

will be interleaving candidates if the conditions are satisfied for any pair of references to the arrays. We perform this analysis on all the important loops of the application, and find pair of arrays, which are interleaving candidates, we take the union of interleaving candidates. Thus if arrays A and B are found to be interleaving candidates from one loop, while B and C are interleaving candidates from some other, then all the three arrays will be interleaved.

Interleaving Arrays

The process of interleaving r arrays of the same data type $A_1, A_2, ...A_r$ is a three step transformation. The first is to replace the individual array declarations with a single array A of r times the size of each array, and second is to fix the access functions of all the array references. The access function $f_m = A_m[a_m i + b_m j + c_m k + d_m]$ of the m^{th} array is replaced by $f_m = A[r \times (a_m i + b_m j + c_m k + d_m) + (m-1)]$ in three-level nested loop. At the end of the day, it is important to schedule the instructions that access the interleaved array in the same pattern consecutively. This is done by moving the result of the first instruction in a new temporary variable, and replacing all its uses by the temporary variable. Interleaving of r arrays of different data types is done by declaring a new structure, say s, which contains an element from each of the arrays. We then declare an array A of the same size as all the previous arrays consisting of elements of data type s. Then we replace the access function of the m^{th} array $f_m = A_m[a_m i + b_m j + c_m k + d_m]$ by $f_m = A[a_m i + b_m j + c_m k + d_m].m$.

Experimental Results

We translate the source code into the FORAY format [50], which essentially consists of just the loop structure and the array access functions as affine functions of the loop iterators. We analyze the code in this format, and, perform our page-aware array interleaving transformations in this format, and then convert it back to the source code. The application is compiled again, and our instruction scheduling for page switch minimization is applied to enhance the impact of array interleaving. Figure 6.7 plots the page switch count after performing array interleaving and instruction scheduling on all the benchmarks. The plot thus shows that our page-aware array interleaving is a very effective transformation, and reduces the data-TLB page-swith count by an average of 35% (indicated by the right-most bar) with an overall average of 11% increase in performance. This performance improvement is inherent to array interleaving, as it inherently increases the spatial locality of data, leading to improved cache behavior. In `swim`, two global arrays and 5 local arrays were accessed together in the loop bodies. Interleaving was possible on all the arrays, thereby forming two interleaved arrays (one global, and other local). This transformation enhanced the opportunities for instruction scheduling and therefore 70% page switch reduction was observed. Since the TLB power is directly proportional to the number of accesses, we can

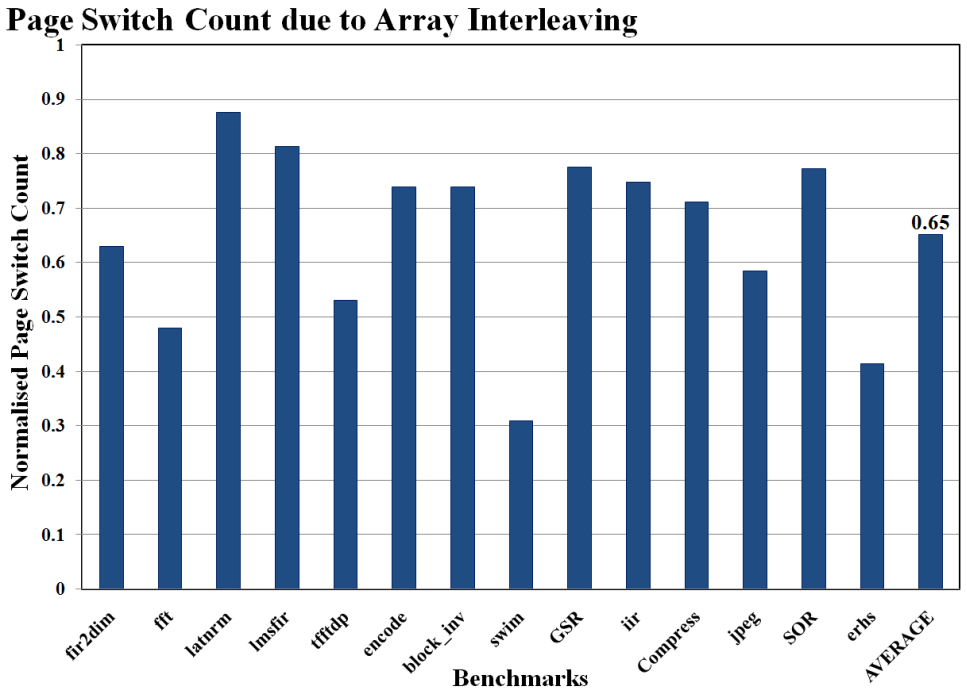


Figure 6.7: Impact of Array Interleaving and Instruction Scheduling on Page Switch Count

expect a concomitant 35% reduction in TLB power due to the combined impact of array interleaving and page switch-aware scheduling.

Impact of Loop Unrolling

Loop unrolling is a loop transformation in which the loop body is replicated a finite number of times, thereby reducing the loop overhead instructions. It is important to observe that loop unrolling by itself does not reduce TLB page switching, but, it may increase the effectiveness of instruction scheduling, by providing more opportunities to schedule instructions and thereby reduce inter-instruction page switching.

Unrolling a loop may reduce page switches if there is at least one instruction, such that if we schedule two copies of the instruction belonging to different iterations when scheduled consecutively, will not result in inter-instruction page switching.

In other words, loop unrolling can be performed if $\exists i \in I$ such that,

$$\begin{cases} min \begin{cases} PS_O(undefined, i.d, i.s1) & \text{if } i.op \text{ is comm} \\ PS_O(undefined, i.d, i.s2) & \end{cases} & = 0 \\ PS_O(undefined, i.d, i.s1) & \text{otherwise} \end{cases}$$

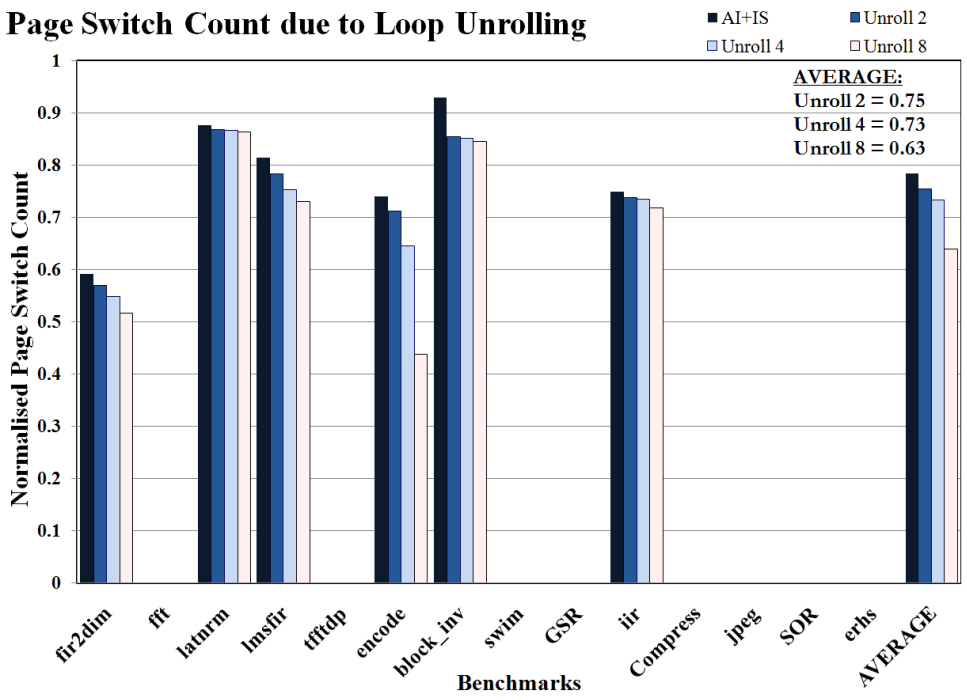


Figure 6.8: Impact of Loop Unrolling on Page Switch Count

We have implemented our page-switch aware loop unrolling transformation also as source code transformation. Figure 6.8 plots the effect of loop unrolling on the page switch count of various benchmark applications. The normalized page-switch count for the case when page-switch aware instruction scheduling and array interleaving are performed is plotted as the dark bar (to the left for each benchmark), and the lighter graphs indicate the page-switch count for unrolling factors of $2, 4$ and 8 times respectively. The right-most set of bars in Figure 6.8 indicate the average values for the cases plotted. On an average, for an unrolling factor of 8, we obtain a reduction of 37% in the page switch count for the applications on which page-aware loop unrolling was possible with 9% performance improvement.

Comprehensive Page Switch Reduction (PSR)

Finally we study the impact of all the three transformations together. The ordering of the transformations is an interesting issue. Instruction scheduling and array interleaving are the fundamental transformations that reduce data TLB page switches. Loop unrolling will be most effective when all the opportunities for page switch reduction achievable after re-scheduling, are exploited. Our page switch-aware instruction scheduling is done at a more fine-grained level, and therefore has to be performed only after array interleaving and unrolling to maximize the effect. We first perform *Page-Switch Aware Array Interleaving* to group the memory allocation of varied arrays together into one overlapped page, and then *Loop Unrolling* on the instructions such that all the instructions capable of being implemented without page-switch are

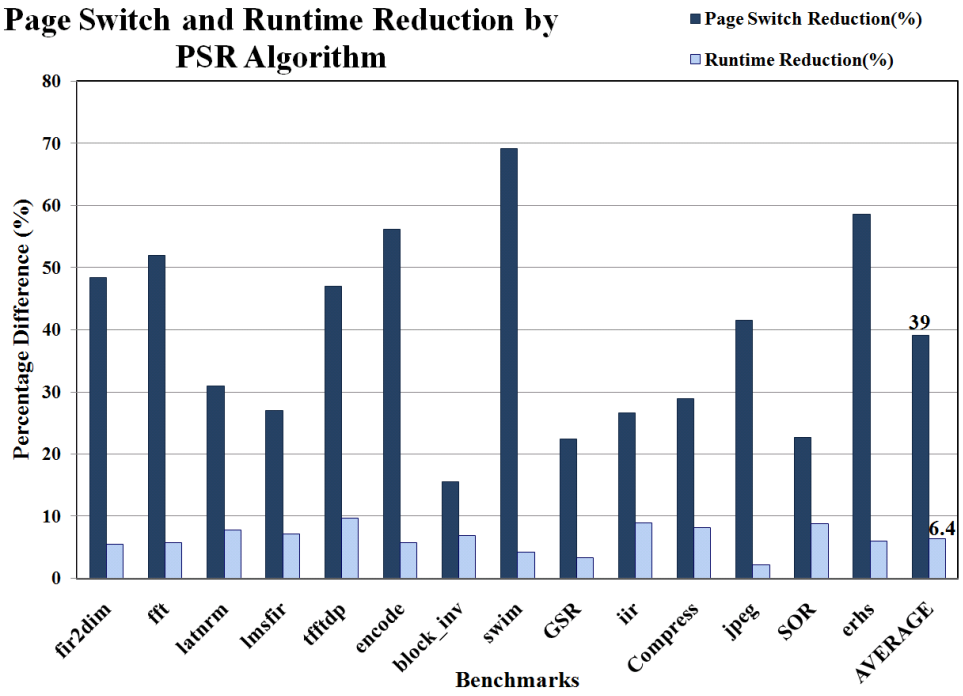


Figure 6.9: Page Switch Count and Runtime reduction by our Page-Switch Reduction Algorithm

executed together. Our fine-grain instruction scheduling is then performed as a post-pass.

The dark bars on the left in Figure 6.9 plot the percentage reduction in the data TLB page switch count for each application. The reduction is calculated as compared to the data TLB page switch count when the application is compiled using $GCC - O3$ alone. The rightmost dark bar shows that there is an average 39% data TLB page switch count reduction over all the benchmarks. The light bars on the right in Figure 6.9 plot the reduction in runtime for all the applications. The rightmost light bar shows that there is an average 6.4% reduction in runtime. In conclusion, the effect of page switch reduction techniques is additive, and the effect is realized after each step of the *Page Switch Reduction* algorithm.

6.3 Instruction-TLB Power Reduction

An important feature of embedded systems is their support of small page sizes. Smaller pages are preferred in embedded systems, as the applications are small, and small page sizes result in better utilization of the limited memory in the embedded system. For example, the ARMv5 [5] and later architectures support the tiny page, in which the page sizes can be as small as 1KB, as compared to the default 4KB. Although tiny pages have a performance benefit, they result in more TLB misses and therefore increase the power consumption of the TLB. The combination of V/P caches and small page sizes (for performance reasons), result in the TLB becoming not only a significant consumer of the processor power budget, but also an important thermal hotspot on the embedded processor. Ekman et al. [30] note that the

TLBs can consume $20-25\%$ of the total $L1$ cache energy. Kadayif et al. [52] find that address translation logic consumes as much as 17% of on-chip power in the Intel StrongARM and 15% in the Hitachi SH-3 processors. In addition, they also find that instructuion-TLB has a power density of 7.820 nW/mm^2, compared to 0.975 and 0.670 nW/mm^2 for $iL1$ and $dL1$ caches, respectively. Reducing the power consumption of TLBs in embedded systems is therefore an important research problem.

Most of the previous research efforts in reducing TLB power consumption were at the hardware level [19, 25, 58, 67]. One effective microarchitectural technique for TLB power reduction, is the *Use-Last* TLB architecture [26, 37], in which the *Use-Last* latch stores the TLB tag of the last translated page. During a program execution, since majority of the cache accesses are to the same page, there will be lesser TLB lookups resulting in power savings. Essentially, power is only consumed in the TLB when consecutive accesses are to different pages. Although the *Use-Last* TLB architecture achieves 75% reduction in i-TLB power, there is scope for further i-TLB power reduction by altering the relative position of the code so as to minimize the total number of page-switches in the program.

In this section, we,

1. formulate the problem of reducing the total number of page-switches in a program, as a code placement problem.
2. prove that the code placement problem for reduced page-switches is NP-complete.
3. propose an efficient ***Bounds Based Procedure Placement (B2P2)*** heuristic for use in the *linker-phase* of the program compilation, to reduce the program's total page-switch count, and thus achieve i-TLB power reduction.

Related Work

Hardware Approaches

At the hardware level, circuit and microarchitectural modifications aim to reduce the per-access power consumption of the TLB and the Memory Management Unit as a whole. Over the years, a fully associative TLB architecture with (Content Addressable Memory) CAM implementation has been proved to be efficient in terms of performance and power. Manne et al. [67] propose a Banked Associative design for TLBs (BA-TLB) which consumes less power than a fully associative TLB. Through the use of a banked cache design, during each access to the TLB, only half the CAM entries are looked up and therefore overall power-per-access is reduced. In another technique, the TLB is constructed as multiple banks with a small filter-bank buffer located above its associated bank [58]. Through the use of selective fil-

tering and banking mechanisms, the number of entries activated on each access is reduced and therefore efficient in embedded processors.

Choi et al. [25] in their work, propose a two-level TLB architecture that integrates a 2-way banked filter TLB with a 2-way banked main TLB design. This architecture, aims at reducing the power consumption of the TLB, by distributing the TLB accesses across the banks in a balanced manner. Chang [19] presents a real-time filter scheme to remove redundant TLB accesses by distinguishing them as soon as the virtual address is generated. This in combination with two adaptive banked TLB designs, has proved to effectively improve the energy delay product of data TLBs. Kadayif et al. [51] introduce Translation Registers (TR) to store the most frequently accessed TLB address translations as a lookup table matching the virtual and physical address tags. During subsequent cache accesses, these TRs are looked up first and if present, no translation is performed (the information stored is used). This saves on switching activity at the register files, mapping the virtual address to their physical address. It should also be noted here that the granularity at which this technique achieves power reduction is influenced by the number of registers or successive access to the architecture blocks. The power savings achieved by such hardware techniques are therefore limited by the area, power and performance traded-offs realized in their implementation.

Software and Hybrid Approaches

At the compiler-architecture interface, the problem of power reduction in the TLB manifests itself as the problem to efficiently reduce accesses on the TLB through optimal changes in the software execution. The key difference between hardware and software approaches is the fact that the TLB architectures are identical for both instruction-TLB and data-TLB, whereas the access patterns of the instruction-cache and data-cache vary significantly. The implementation and design of a *software technique* for TLB power reduction, varies according to the targeted TLB structure. On the other hand, a *hybrid approach* has the critical advantage of proposing architectural modifications and corresponding software techniques that make efficient use of the underlying architecture, achieving efficient results. The state-of-the art software and hardware approaches can be broadly classified based on their target TLB structure.

Software Techniques

Parikh et al. [80] propose a set of energy-oriented instruction scheduling techniques where the instructions within a basic-block of code is scheduled with regard to its energy consumption. The energy component is calculated as a weighted cost function: `circuit-state-cost` for each schedule of instructions. Energy-oriented scheduling achieves 30% reduction in energy as compared to performance-oriented scheduling. Chiyonobu et al. [24] in their work propose an efficient scheduling technique that

allows for the execution of critical instructions on power-hungry functional units, and the other instructions on power-optimal units, thereby reducing the overall power consumption of the system. This scheduling technique achieves an average 27.3% ED^2P reduction with 1.4% performance degradation. It should be noted here that the impact of these software techniques on a broad spectrum of applications are limited by the underlying architecture and also realize a performance trade-off. As far as our knowledge goes, no software only approach has been proposed for instruction-TLB power reduction.

Hybrid Approaches for data-TLB

A compiler-directed array interleaving technique was proposed to save energy in multi-bank memory architectures with power control features [27]. In this, the arrays used in separate banks are interleaved, such that only one of the banks is active and the other can be powered down, thus saving energy. Though effective in power savings, the energy reduction achieved by this technique does not account for the leakage power of the SRAM cells during standby mode for current and future technology embedded processors. Kandemir et al. [53] propose to increase the effectiveness of *Translation Registers* (TRs) to reduce the data-TLB power consumption through compiler optimizations (using profile information) to maximize reuse of the data stored in the TRs. This technique incurs a performance overhead of 3.5% due to compiler updates and achieves an average of 32.6% reduction in TLB lookups. In addition, the proposed technique requires changes to the (Instruction Set Architecture) ISA, which may not be desirable for many embedded applications. In Section 6.2, we develop a static compiler technique to achieve data-TLB power reduction, effectively utilizing the *Use-Last* TLB architecture implementation. In this, we present a series of page-aware code transformations (instruction re-ordering, array interleaving and code fission/fusion), and an all-inclusive comprehensive algorithm that demonstrates an average of 39% data-TLB power reduction with negligible impact on performance. These software and hybrid techniques proposed, target the data cache and data-TLB accesses only. Owing to the significant difference in data and instruction cache access patterns, their effectiveness is restricted to data-TLBs. Our Bounds Based Procedure Placement (B2P2) heuristic, accentuates the applicability of this *Use-Last* TLB architecture and thus achieves maximum possible i-TLB power reduction on a wide range of applications. We assume henceforth that the underlying embedded processor used for our description and analysis has an implementation of the *Use-Last* TLB architecture for the i-TLB structure, and our objective is to reduce the number of page-switches that occur during program execution.

Hybrid Approaches for instruction-TLB

One effective hybrid approach with the goal to reduce instruction-TLB power, is by Kadayif et al. [52], where they propose a set of software only, hardware only and integrated hardware-software techniques.

In this, the processor is facilitated with a set of Translation Registers (TRs) that assist in storing recently accessed page translations. The compiler techniques proposed, aim to reduce the instruction TLB lookups by changing the i-cache access patterns, by introducing marker instructions for intelligent use of the TRs. This technique achieves 85% i-TLB power savings and proves to be effective only for larger and slower i-TLB structures. Our work though simlar in intent, differs from this based on the underlying TLB architecture. In [52], an array of power-hungry registers are used to maintain a lookup table, while our work involves the implementation of an energy efficient *Use-Last* TLB architecture [37], involving only limited hardware additions. Again, the size and design of these registers (TR) has a significant impact on the effectiveness of their technique (work in [52]), while the key component of the *Use-Last* TLB is a latch (detailed discussion of the architecture is available in Section 6.2).

In the previous sectionSection 6.2, we perform code transformations to efficiently utilize the *Use-Last* TLB architecture implementation, and reduce data-TLB power consumption. In this work, we propose a similar compiler-microarchitecture hybrid approach over the *Use-Last* TLB architecture, to reduce instruction-TLB lookups and thereby power. This being an optimization technique over the instruction cache accesses, the program's profile information is used as input to the *B2P2 heuristic*. The result of this heuristic are the start addresses of the procedures in the program, optimized for intelligent page-locality and when executed on the *Use-Last* TLB architecture, achieve reduced i-TLB power consumption.

Page-Switch Reduction by code Placement
Page-Switches in the Instruction Memory

In any program, the total number of page-switches incurred can be classified as follows:

1. Function-call Page-Switches (PS_F): The set of page-switches in a program, caused due to function-calls executed across a page-boundary are called *function-call page-switches*, denoted by PS_F.

2. Loop-execution Page-Switches (PS_L): The page-switches incurred during the execution of loops that span across page-boundaries, are called *loop-execution page-switches*, denoted by PS_L.

3. Sequential-execution Page-Switch (PS_S): The page-switches caused during sequential instruction execution within the basic-blocks of the program, are called *successive-access page-switches*, denoted by PS_S.

4. The total number of page-switches in the program is thus given by: $TPS = PS_F + PS_L + PS_S$.

Objectives for Page-Switch Reduction

To minimize the page-switches caused during program execution, the required modifications on the code fall under one of the following cases based on the type of instructions involved:

1. The call-site and the start address of the callee-function should reside in the same page, to avoid page-switches during the function-call.
2. The call-site and the end address of the callee-function should reside in the same page, to avoid page-switches during the call-return.
3. For loops of size atmost page-size, the loop should be positioned to completely reside in a single page and avoid page-switches on each iteration.
4. For loops of size at least page-size, the loop has to be positioned to span across minimum number of page-boundaries as possible.
5. The functions of size atmost page-size, should be positioned completely within a page to remove the page-switches incurred during each function-call.
6. For functions of size greater than a page-size, the function has to be positioned such that it spans across minimal number of page-boundaries.

Granularity of Code Placement for Page-Switch Reduction

Compiler directed code placement techniques can alter the relative position of the instructions in the program and thus vary their instruction memory access patterns. At the compiler, this problem of page-switch reduction can be approached at different granularities: (i) Instruction level, (ii) Basic-block level and (iii) Procedure level.

Instruction Level Granularity

At the instruction level granularity, code placement involves reallocating instructions or a set of instructions in the memory, while their original control sequence is maintained with the help of inserted control instructions (branch, jump, jump-and-link,etc.). This fine granularity of approaching the code placement problem gives greater freedom for reallocation and probably maximum page-switch reduction in the program. This technique involves the insertion of control instructions and also variable number of `nop` instructions for page-alignment purposes. Addition of these instructions have the following disadvantages:

- Increase in code-size due to the inserted instructions may be of concern for embedded applications.
- The added executable instructions (branch, jump, jump-and-link, etc.), increase the runtime of the application and thereby affect the performance of the system. In the presence of `nop` instructions, out-of-order scheduling of instructions on a multi-issue processor affects the overall performance of the system.
- The branch and jump instructions added, activate the branch-target buffer and allied branching hardware thus increasing the accesses to such power-hungry components of the processor. The overall power reduction achieved through any optimal code placement, could thus be overthrown by the increase in runtime and overall power consumption of the system.

Basic-block Level Granularity

At the basic-block level, existing branch instructions between blocks can be reassigned to a new address when reallocated, but new control instructions will have to be inserted to maintain the control of the *fall-through* basic-block. Therefore, comparatively significant number of instructions are required to be added to the code. Here again, variable number of `nop` instructions may be added for page-alignment purposes. Approaching the problem at a more coarser granularity causes lesser freedom for movement of the code and thus may lead to lesser page-switch reduction than that at the instruction level. The disadvantages that plague instruction-level code placement (described above), also impact basic-block level code placement, but to a relatively lesser degree.

Procedure Level Granularity

At an even more coarser granularity, the procedure blocks can be reallocated in the instruction memory. No control instructions are required for this modification as the procedures already have branch instructions for the program control and only the target addresses have to be varied accordingly. Owing to the coarser granularity, freedom to move the code blocks is restricted and therefore the possible page-switch reduction is relatively lesser. Since the TLB structure is a small part of the processor, any power reduction technique for the TLB should consider its impact over the system power as a whole and therefore additional instruction insertions should be avoided.

In this work, we formulate the code placement problem for minimized page-switches, at the procedure level granularity and define it as a *Procedure Placement Problem* (PPP). In this, the functions[1] in the program are moved as a whole. No executable instruction is introduced into the existing program code, and padding (if any) for page alignment, is done by using `nop` functions. The challenge here is to

[1] We use the words *function* and *procedure*, interchangeably to denote procedure blocks of a program.

efficiently place the procedures in the instruction memory, such that the total number of `nop` functions added are minimized and page-switches incurred are minimal. This mechanism experiences a variation in the overall program runtime, only due to the instruction cache associativity factors. We observe through experiments that this performance variation is limited to less than 2%.

The Procedure Placement Problem (PPP)

The problem here is to assign start addresses to the functions in a program such that, the program execution incurs reduced number of page-switches and thereby reduced i-TLB power.

Input

The program can be represented by a hierarchical structure of tuples rooted at P. The tuple $P = <n, FN[]>$ lists the set of $P.n$ functions, in the form of a tuple array $P.FN[]$, where each entry is represented by the 6-tuple $FN_x = <Id, Pos, Size, Calls, CS[], LP[]>$. In this, $FN_x.Size$ represents the function size, $FN_x.Id$ the unique function-id and $FN_x.Calls$ the total number of calls to the function. The set of call-sites and loops within the function are represented by their respective tuple arrays $FN_x.CS[]$ and $FN_x.LP[]$. Each is a 4-tuple described as follows.

Call-site tuple $FN_x.CS = <Id, Offset, Callee, Count>$ within function FN_x:

- $CS_i.Id \leftarrow$ id of the call-site in the function FN_x.
- $CS_i.Offset \leftarrow$ represents the position of the call-site from the start of the function.
- $CS_i.Callee \leftarrow$ contains the callee function-id (e.g., FN_y).
- $CS_i.Count \leftarrow$ indicates the number of calls to the callee function FN_y from FN_x.

Loop tuple $FN_x.LP = <Id, Offset, Size, Count>$ within function FN_x:

- $LP_j.Id \leftarrow$ id of the loop in the function FN_x.
- $LP_j.Offset \leftarrow$ represents the position of the loop start address from the start of the function.
- $LP_j.Size \leftarrow$ represents the size of the loop in bytes.
- $LP_j.Count \leftarrow$ indicates the total number of iterations of the loop.

Output and Constraint

The output of our procedure placement problem are the values $FN_x.Pos \; \forall FN_x \in P$, that represent the start addresses of the functions, under the constraint that no two functions should overlap each other in the instruction memory.

$$PS_F = \sum_{\forall x: FN_x \in P} \sum_{\forall i: CS_i \in FN_x.CS[\,]} (FPS_F(FN_x.CS_i) + RPS_F(FN_x.CS_i)) \times CS_i.Count \quad (6.2)$$

$$PS_L = \sum_{\forall x: FN_x \in P} \sum_{\forall i: LP_i \in FN_x.LP[\,]} (FPS_L(LP_i) + RPS_F(LP_i)) \times LP_i.Count \quad (6.3)$$

$$PS_S = \sum_{\forall x: FN_x \in P} FN_x.Calls \times LonePB(FN_x) \quad (6.4)$$

Objective

Given a program, the procedure placement problem can be defined as the problem to relatively position the functions in the instruction memory (assignment of start addresses $FN_x.Pos$) such that page-switches caused by loops (PS_L) or functions (PS_S) crossing page boundaries during their execution or function calls (PS_F) to callee-functions on different pages, are minimum. Given a program and its profile information in the form of the tuple hierarchy (defined above), the objective to minimize the total number of page-switches is given by Equation (6.1), where the individual components are represented by the equations: Equation (6.2)($PS_F(FN_x)$), Equation (6.3) ($PS_L(FN_x)$) and Equation (6.4) ($PS_S(FN_x)$).

$$minimize \sum_{\forall FN_x \in P} PS_F(FN_x) + PS_L(FN_x) + PS_S(FN_x) \quad (6.1)$$

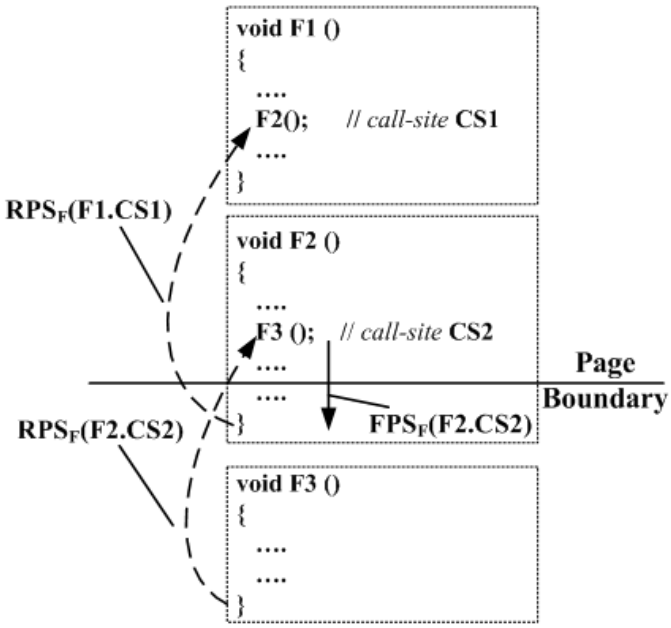


Figure 6.10: ***Function-call Page-Switches*:**
(i)$PS_F(F2)$ is the sum of the function's caller-to-callee page-switches ($FP_F(F2.C2)$) and callee-to-caller function-return page-switches($RPS_F(F2.C2)$).
(ii)$PS_F(F1)$ is the number of callee-to-caller function return page-switches($RPS_F(F1.C1)$).

PS_F: Page-Switches due to Function-calls

The total number of page-switches in the function FN_x, due to function calls at the call-site $FN_x.CS_i$, is equal to the sum of *forward page-switches* (FPS_F) and *reverse page-switches* (RPS_F). In Figure 6.10, the call-site $CS1$ in function $F1$ experiences a page-switch only during the return from $F2$, since only the end address of $F2$ is on a different page. In the case of call-site $CS2$ in function $F2$, since the function $F3$ entirely resides in a different page, both the function call and function return experience forward and reverse page-switches respectively. The total number of page-switches in a program due to function calls, is given by Equation (6.2).

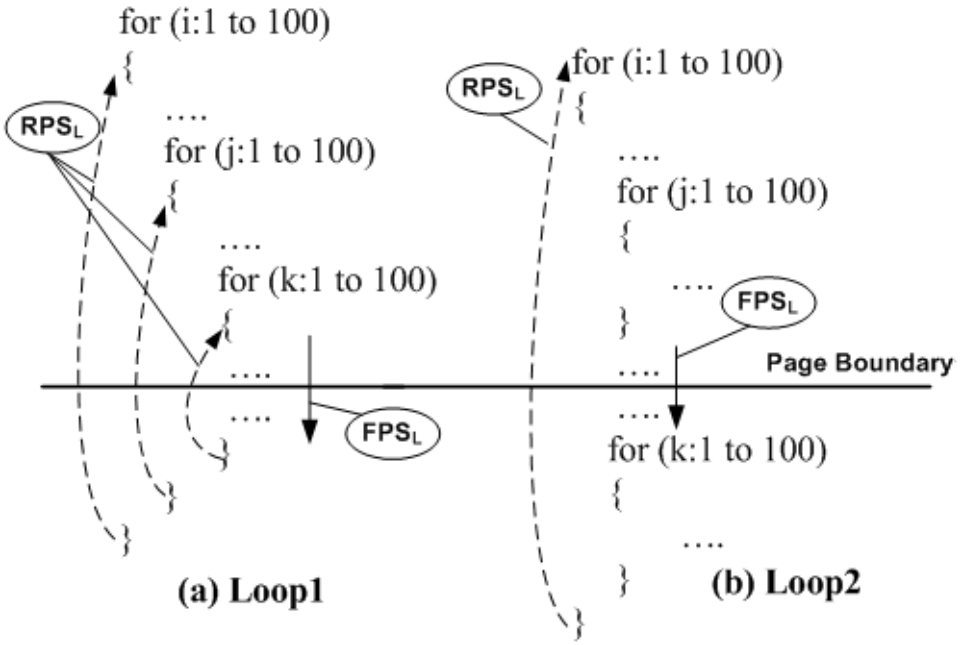


Figure 6.11: ***Loop-execution Page-Switches*** **for:**
(a) $PS_L(Loop1)$ is equal to the sum of its forward iteration $FPS_L = 100^3$ and loop-return $RPS_L = (100) + (100^2) + (100^3)$.
(b) $PS_L(Loop2)$ is the sum of $FPS_L = 100$ and $RPS_L = 100$.

PS_L: Page-Switches due to Loop iterations

The total number of page-switches in the function FN_x, due to loops that span across page boundaries, is the sum of *forward page-switches* (FPS_L) and *reverse page-switches* (RPS_L). We will describe the nature of these page-switches through examples. In the nested loop structure as in $Loop1$ of Figure 6.11(a), the instruction accesses cross a page-boundary only for the innermost loop (indicated by solid arrow) in the forward direction, while the last and first instructions of every loop is accessed in the reverse direction. Thus for this example, the total number of page-switches is given by $FPS_L = 100 \times 100 \times 100$ (total iteration count of the innermost loop) and $RPS_L = (100) + (100 \times 100) + (100 \times 100 \times 100)$. In the case

of a loop structure as in *Loop*2 of Figure 6.11(b), the page-boundary is crossed during each iteration of the outer loop (over i) and not by any of the inner loops. Therefore, the total page-switches is given by $FPS_L = 100$ and $RPS_L = 100$. The total page-switches due to loops in the program is thus given by Equation (6.3).

PS_S: Page-Switches due to Sequential Accesses in functions

A page-switch is incurred when a basic block, not covered by a loop within the function, spans across a page-boundary. The total number of page-switches PS_S, incurred by such basic blocks, for each function is equal to the product of the function call-count $FN_x.Calls$ and the number of such *lone page-boundaries* crossed within the function. If function $LonePB() : FN_x \rightarrow N$ return the number of lone page-boundaries within each function $FN_x \in P$, the total page-switches within function blocks in the program is given by Equation (6.4).

Intractability of the Procedure Placement Problem

In deriving the computational complexity of the PPP, we take a subset of the problem and prove that this problem-subset, obtained by adding constraints on the input of the PPP, is NP-complete and therefore our PPP is definitely NP-complete.

Subset of the Problem: PPPS

In order to prove the intractability of our problem, we derive here a subset of the problem and use that in our reduction from a known NP-complete problem. The input to this problem subset is restricted in the sense that only the page-switches due to function-calls are considered and so the function is described by a set of call-sites and their corresponding call-counts. The placement of the functions into pages is constrained, such that a function can be placed in a page if and only if the whole function fits into the page (i.e., a function cannot reside in two pages).

Decision Version of PPPS

Let us consider a program with n functions (**F**) and p pages (**P**) available for allocation. The size of each function is denoted by $w(f)$ for each function $f \in \mathbf{F}$. A caller-to-callee function-call is denoted by the calls $c \in \mathbf{C}$ that connects the two functions. The function call-count between the two functions is given by the cost function $t(c)$ for each call $c \in C$. The size of each page is a constant given by **S** and an upper bound equal to the maximum page-switch cost $\mathbf{M} \in \mathbb{Z}^+$ for the program. This is formally defined as follows:

INSTANCE: Set of functions **F** and edges **E**, function sizes $w(f) \in \mathbb{Z}^+$, $\forall f \in \mathbf{F}$, page-switch costs $t(c) \in \mathbb{Z}^+$, $\forall c \in \mathbf{C}$, page-size constant $\mathbf{S} \in \mathbb{Z}^+$ and upper bound on page-switch cost $\mathbf{M} \in \mathbb{Z}^+$.

QUESTION: Is there a partition of the functions $\mathbf{F}$ into p disjoint subsets $\mathbf{F_1}, \mathbf{F_2}, \cdots, \mathbf{F_p}$, such that $\sum_{f \in \mathbf{F_i}} w(f) \leq \mathbf{S},\ 1 \leq i \leq p$, and for all calls $\mathbf{C}' \subseteq \mathbf{C}$ that have their caller and callee functions on two different subsets F_i, F_j, then $\sum_{c \in \mathbf{C}'} t(c) \leq \mathbf{M}$?

The Graph Partitioning Problem (GPP) [32]

Given a graph $G = (V, E)$, where $w(v)$ defines the weight of each vertex $v \in \mathbf{V}$, and each edge $e \in \mathbf{E}$ has a cost $c(e)$ attached to it. Two positive integers defined are $\mathbf{K}$ and $\mathbf{J}$. This can formally be defined as follows:

INSTANCE: Graph $G = (V, E)$, weights $w(v) \in \mathbb{Z}^+,\ \forall v \in \mathbf{V}$, and cost $c(e) \in \mathbb{Z}^+,\ \forall e \in \mathbf{E}$, positive integers $\mathbf{K}$, $\mathbf{J}$.

QUESTION: Is there a partition of $\mathbf{V}$ into disjoint sets $\mathbf{V_1}, \mathbf{V_2}, \cdots, \mathbf{V_m}$, such that $\sum_{v \in \mathbf{V_i}} w(v) \leq \mathbf{K},\ 1 \leq i \leq m$, and such that if $\mathbf{E}' \subseteq \mathbf{E}$ that have their two endpoints in two different subsets V_i, V_j, then $\sum_{e \in \mathbf{E}'} c(e) \leq \mathbf{J}$?

The Reduction: GPP $\leq_p$ PPPS

From an instance of the GPP, an instance of PPPS can be generated in polynomial time as follows:

- Graph $G = (\mathbf{V}, \mathbf{E})$ in GPP $\Rightarrow$ program call-graph $(\mathbf{F}, \mathbf{C})$ formed of functions and function-calls.
- vertices $\mathbf{V}$ in GPP $\Rightarrow$ set of functions $\mathbf{F}$ in PPPS.
- edges $\mathbf{E}$ in GPP $\Rightarrow$ set of function-calls $\mathbf{C}$ in PPPS, where the end-points indicate the caller and callee functions.
- weight of vertex $w(v)$ in GPP $\Rightarrow$ function size $w(f)$ in PPPS.
- cost of edge $c(e)$ in GPP $\Rightarrow$ page-switch cost due to function-call $t(c)$ in PPPS.
- The m partitions of $\mathbf{V}$ in GPP $\Rightarrow$ the p pages into which the program's functions have to be allocated.
- The disjoint subsets $\mathbf{V_1}, \mathbf{V_2}, \cdots, \mathbf{V_m}$ in GPP $\Rightarrow$ the subset of functions $\mathbf{F_1}, \mathbf{F_2}, \cdots, \mathbf{F_p}$ allocated in the p pages.
- Constraint $\sum_{v \in \mathbf{V_i}} w(v) \leq \mathbf{K} \Rightarrow \sum_{f \in \mathbf{F_i}} w(f) \leq \mathbf{S}$ that defines the upper bound on the functions that are allocated to a page.
- Objective $\sum_{e \in \mathbf{E}'} c(e) \leq \mathbf{J}$ in GPP $\Rightarrow \sum_{c \in \mathbf{C}'} t(c) \leq \mathbf{M}$, that defines the minimization function with an upper bound on the page-switch cost.

Proof

A given solution to the PPPS problem can be verified in polynomial time, and therefore we deduce that the PPPS is of NP class. From the reduction above, we observe that an instance of GPP can be directly converted into an instance of the PPPS, from the call-graph of the program. Therefore, we derive that an optimal solution to the PPPS exists if and only if there exists an optimal partitioning of the vertices in GPP. Given an instance of the GPP, a `YES` decision on the partitioning Π, indicates that there exists a partition of the functions Δ for an equivalent instance of the PPPS. Conversely, if we obtain a `YES` decision for the partition Δ on an instance of the PPPS, we can say that there exists a partitioning Π for an equivalent instance in GPP.

The decision version of GPP, is a known NP-complete problem [32]. Having reduced an instance of GPP to PPPS (GPP $\leq_p$ PPPS), we have shown that the PPPS is atleast as hard as GPP and therefore definitely NP-complete. The PPPS (from our description above) is a subset of our original code placement problem PPP with limited constraints in placement and input. From our construction of the PPPS problem, and nature of the constraints, we deduce that *the problem to obtain an optimal solution to our procedure placement problem for minimized page-switches is NP-complete*.

Related Code Placement Techniques

Over the years, researchers have developed various code placement techniques targeting power and performance issues in embedded processors. Xianglong et al. [43] develop a dynamic code management technique for processors with managed runtimes, reducing the total number of i-TLB misses and thereby improving performance. Here, a JIT compiler is used to analyze the program call graph, and reallocate procedures with high call frequency closer together. Researchers in [17, 100] propose different compiler optimizations and code placement techniques to increase code locality and reduce cache-misses, efficiently improving performance of the embedded system. The authors in [6, 52] propose to enhance cache and TLB effectiveness through compiler-directed code transformation techniques. Bernhard et al. [29] propose a dynamic code placement technique for i-cache power savings, where the instruction cache is replaced with a scratchpad and mini-cache. Program profile information is used to map high execution blocks (loops) to the scratchpad (fixed size), thereby achieving power savings. The runtime overhead and code size increase due to instruction insertions (as discussed in Section 6.3) is compensated by the energy and performance advantage of scratchpad memories.

Though our procedure placement problem resembles in principle with some of the above compiler techniques, it differs from them in the problem formulation and underlying architectural intricacies.

We target embedded processors, and propose compiler techniques to support architectural modifications that can be implemented in a wide variety of applications. In techniques that increase code locality by reallocating procedures with high call frequency closer together, it should be noted that the existence of loops within the procedures and their respective iteration counts are not considered. In a program, there may exist a case where majority of the execution time is spent on a loop within a procedure that has only one call to it, and when such a loop crosses a page-boundary, page-switches are incurred. In our B2P2 technique, we profile the loops (iteration count) and procedures (call count) to efficiently position the procedures, such that minimal page-switches are incurred during the execution of such loops/procedures. During procedure placement, to reduce power and runtime overheads, the entire procedure is treated as one entity and positioned in relation to the page-boundaries, provided no executable instruction is inserted into the code.

ILP Formulation of a PPP Problem-Subset

Having proved that PPPS is NP-complete in Section 6.3, we formulate a 0-1 ILP for the problem.

Preliminaries

In accordance with our assumptions and definition of the PPP problem-subset (as described above), the constraints on the tuple variables (defined above) are defined as follows:

- the function size $FN_x.Size \leq PS$.
- the function contains no loops and therefore the tuple array $FN_x.LP[]=\phi$.
- the function contains only function-calls and therefore the tuple array $FN_x.LP[] \neq \phi$.

The notations involved in the *ILP* formulation of the above stated problem subset are defined as follows:

- set of functions $F = \{f_i : 0 \leq i < n\}$
- set of i-cache pages $P = \{p_j : 0 \leq j < n\}$
- function $SizeOf : F \rightarrow N$ that returns the size of function f_i
- $cost_{i1,i2}$ = page-switch cost when function f_{i1} and f_{i2} are in separate pages.
- vector $X = (x_{0,0}, ...x_{0,m}, ...x_{n,m})$ where element $x_{i,j}$ is defined as,

$$x_{i,j} = \begin{cases} 1 & \text{when function } f_i \text{ is allocated in page } p_j \\ 0 & \text{otherwise} \end{cases}$$

Constraints

The constraint equations defining the problem formulation, follow from our assumptions (as described above) and the requirements of an optimal procedure placement for reduced page-switch count. Equation (6.5) defines the constraint that a function can be allocated to only one cache page. Equation (6.6) states that the set of functions allocated to a single page have to fit completely within the page allocated.

$$Page\_Limit_j = \sum_{i:f_i \in F} x_{i,j} \times size_i \leq PS,\ \forall js.t.p_j \in P \tag{6.5}$$

$$Function\_Limit_i = \sum_{j:p_j \in P} x_{i,j} = 1,\ \forall is.t.f_i \in F \tag{6.6}$$

Objective function

The objective function for our problem subset is defined: *"Given the sets F,P, function-sizes SizeOf*$(f_i \in F)$*, and the* $2-D$ *data table containing cost values for all pairs of functions (*$f_x, f_y \in F$*), find the vector X that defines the placement of the functions (*$f_i \in F$*) to pages (*$p_j \in P$*) such that the total page-switch cost is minimized."* This objective function is thus described by the relation:

$$\sum_{\forall i1,i2 \in F} \sum_{\forall p_j \in P} \left(x_{i1,j}(1-x_{i2,j}) + x_{i2,j}(1-x_{i1,j})\right) \times cost_{i1,i2} \tag{6.7}$$

The expression (within the summation) in Equation (6.7), defines the *decision-value*$= 0$, if the functions f_{i1} and f_{i2} can be placed in a single page ($p_j \in P$), and 1 otherwise. The sum of the *decision-value* and the *cost* incurred due to their placement on different pages, over all possible placement options, determines the total page-switch count of the program. The non-linear terms in Equation (6.7) forming the objective function can be transformed into a linearized implementation using well known linearization techniques [99].

We have implemented the above ILP formulation in GMPL and have been able to achieve an optimal solution for reasonable small number of functions and components using GLPK [66] solver. For the call graph similar to that of *dijkstra* benchmark considering only the page-switches due to function-calls, an optimal solution was achieved after 1548 iterations, using 1.3 MB of memory and 661 variables. This formulation only involves one of the constraints that defines the actual code placement problem, and adding further constraints may increase the runtime and memory requirements. This essentially makes this solution impractical for use in a compiler and clearly a more efficient technique is required.

B2P2:Bounds Based Procedure Placement Heuristic

We develop here an efficient *Bounds Based Procedure Placement* (B2P2) heuristic solution to the PPP for minimizing total number of page-switches in a program.

Overview

The profile information gathered from the program, populated into the tuple hierarchy P is the input to our B2P2 heuristic. The *program-elements*:
call-sites ($FN_x.CS_i, \forall i : FN_x.CS_i \in FN, \forall x : FN_x \in P$)
and loops ($FN_y.LP_j, \forall j : FN_y.CS_j \in FN, \forall y : FN_y \in P$)
of the program are listed in the list ELEMENTS_LIST, in the decreasing order of their weights equal to the page-switches incurred during their execution. The weight for a call-site is its corresponding function call-count and that for a loop is its iteration count. Each element from this list is considered greedily (by extracting from top of the sorted heap), for optimal procedure placement, by forming bounds that contain the element within page-boundaries and thereby avoid the occurrence of page-switches during their execution. The formed bounds are affine inequalities over the variable $FN_x.Pos$ (function start-address) of the function FN_x that contains the element under consideration.

During each iteration over the ELEMENTS_LIST, an element (or function) can be assigned to a page *iff*, the newly formed bound does not conflict with any existing bounds for that page. Once bounds are formed for all the elements in the list, the function start addresses $FN_x.Pos$ for the functions assigned to a page is obtained by taking the smallest integer value that satisfies all the inequalities for the corresponding page. By taking only the smallest possible value for the function start addresses, we guarantee that the amount of required padding (using *nop* functions) is minimized. Algorithm 3 describes the main function ALLOCATE_FUNCTIONS() of the heuristic and the following sections describe the implementation details of the ASSIGN_BOUNDS_LP() and ASSIGN_BOUNDS_CS() functions.

Illustration

Input to the Heuristic and DCFG Formation

In order to facilitate the implementation of our B2P2 heuristic, we define a back-pointer in the call-site and loop tuples that represent the functions the element belongs to. The value $FN.CS.FnId$ in the tuple $FN.CS$ represents a pointer to the caller function ($FN.Id$) and $FN.LP.FnId$ in the tuple $FN.CS$ represents a pointer to the function in which the loop is located. Since the input to the heuristic is a call graph (annotated with tuple information from profile data) without any page assignments, the variable $FN.Pos$ is treated as a variable throughout the operation of the heuristic.

Algorithm 3: ALLOCATE_FUNCTIONS(Function_List, Page_List, Elements_List)

Require: Function_List=$(FP_1, FP_2, \cdots, FP_n)$, Page_List=$(PG_1, PG_2, \cdots, PG_n)$,
Elements_List=$(CL_1, CL_2, \cdots, CL_n)$.

```
1: for each element CP_x from Top (ELEMENTS_LIST) do
2:     type ← Get_Element_Type_Of (CP_i)
3:     if type == LOOP then
4:         LP_i ← addr(CP_x)
5:         ASSIGN_BOUNDS_LP (LP_i.FnId, LP_i)
6:     else if type == CALL-SITE then
7:         CS_i ← *CP_x
8:         ASSIGN_BOUNDS_CS (CS_i.FnId, CS_i)
9:     end if
10: end for
```

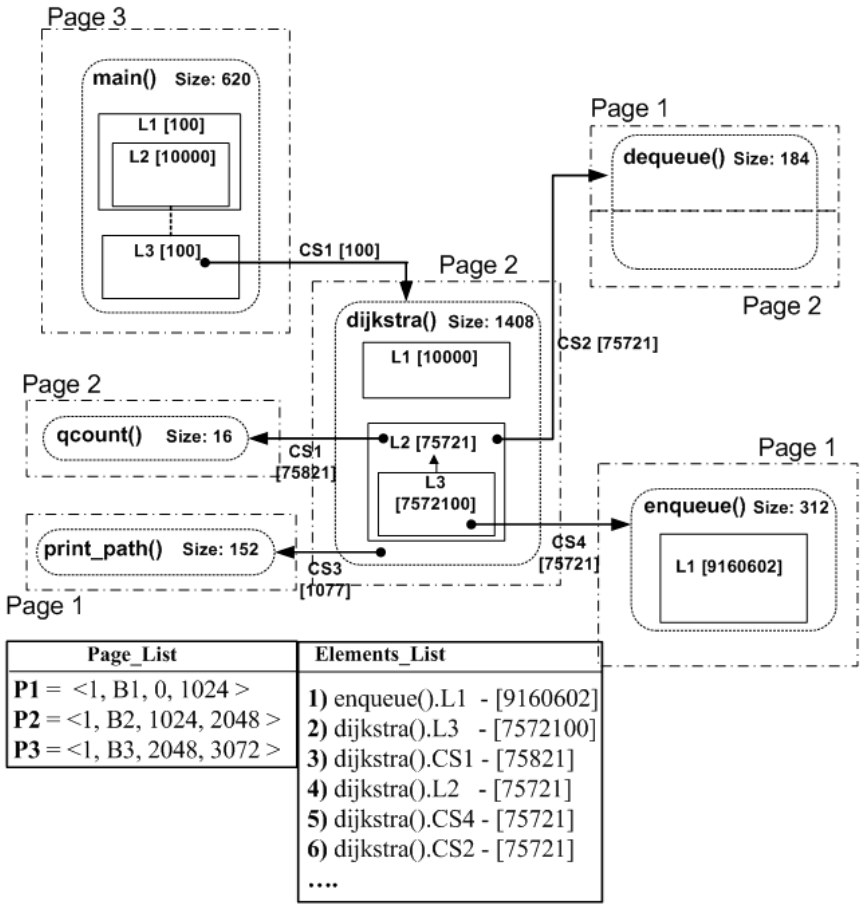


Figure 6.12: Original DCFG of *dijkstra* program with page demarcations.

Figure 6.12 describes the *DCFG* formed for the *dijkstra* benchmark program annotated with the gathered profile information. Here, the rounded rectangles denote the functions in the program. Each program-element: call-site (solid arrows) and loops (solid line rectangles) are annotated with their *id* values and corresponding weights, as indicated in Fig. 4. The page boundaries are marked by dotted lines labeled with their respective page numbers.

The list of pages (PAGE_LIST) available for allocation is described in Fig. 4. The size of this list given by: $\lceil \frac{ProgramSize}{PageSize} \rceil$, and each element in the list is a 4-tuple $PG =< Id, Bounds, SA, EA >$ where, $PG_r.Id$ is the page number, $PG_r.SA$ its start-address and $PG_r.EA$ its end-address. $PG_r.Bounds$ is a vector array initialized to null and used to hold the affine bounds for each page. For each iteration of the loop in Algorithm 3, the program element with highest weight is taken, and its corresponding bounds assignment function (ASSIGN_BOUNDS_LP() or ASSIGN_BOUNDS_CS()) is executed.

Function ASSIGN_BOUNDS_LP()

When the element considered for placement is a loop, affine equations are formed with the loop's start address ($FN_x.Pos + FN_x.LP_i.Offset$), size ($FN_x.LP_i.Size$), and page boundaries ($PG_r.SA, PG_r.EA$) of the available page. The objective is to ensure that the entire loop exists within the page assigned and/or crosses minimum number of page-boundaries as possible.

$$\begin{aligned} LP_i.SA &\leftarrow FN_x.Pos + FN_x.LP_i.Offset \\ LP_i.EA &\leftarrow FN_x.LP_i.SA + FN_x.LP_i.Size \\ bound &\leftarrow PG_r.EA \geq LP_i.EA > LP_i.SA \geq PG_r.SA \end{aligned} \tag{6.8}$$

For the example in Figure 6.12, loop $L1$ of function *enqueue()* is the first element in the ELEMENTS_LIST and the bounds to contain this loop within a page is formed while assigning this function to page $PG_1 =< 1, B1, 0, 1024 >$ and input to the bounds $PG_1.B1$ Equation (6.9). Here, the loop $enqueue().L1$ is of size 24 bytes located at an offset of 208 bytes from the function start address $enqueue().Pos$ obtained by substituting variables in Equation (6.8). When a bound is formed for the next element in the list $dijkstra().L3$, we realize the conflict with the existing bound in $PG_1.B1$ owing to the function size ($dijkstra().Size = 1408$ bytes), and therefore a new bound $PG_2.B2$ Equation (6.10) is formed. In Equation (6.9) and Equation (6.10), the $\leftarrow$ indicates that the newly formed bound is appended with the existing set of bounds.

$$\begin{aligned} PG_1.B1 &= 1024 \geq enqueue().Pos + 208 + 24 \\ PG_1.B1 &\leftarrow PG_1.B1 > enqueue().Pos + 208 > 0 \end{aligned} \tag{6.9}$$

$$\begin{aligned} PG_2.B2 &= 2048 \geq dijkstra().Pos + 1032 + 16 \\ PG_2.B2 &\leftarrow PG_2.B2 > dijkstra().Pos + 1032 > 1024 \end{aligned} \tag{6.10}$$

Function ASSIGN_BOUNDS_CS()

When the element considered is a call-site, affine equations are formed with the call-site position $CS_i.Addr$, the callee-function start address ($Callee.Pos$) and size ($Callee.Size$), within page boundaries ($PG_r.SA, PG_r.EA$) of an available page. The objective is to ensure that both the callee function and the call-site (at the caller function) are in the same page.

$$\begin{aligned}
&CS.Addr \leftarrow FN_x.Pos + FN_x.CS_i.Offset \\
&Callee.Pos \leftarrow (FN_x.CS_i.Callee).Pos \\
&Callee.EA \leftarrow Callee.Pos + (FN_x.CS_i.Callee).Size \\
&tb1 \leftarrow PG_i.EA \geq CS.Addr \geq PG_i.SA \\
&tb2 \leftarrow PG_i.EA \geq Callee.EA \geq Callee.Pos \geq PG_i.SA \\
&bound \leftarrow tb1 \ \& \ tb2
\end{aligned} \tag{6.11}$$

In Figure 6.12, the third element on the ELEMENTS_LIST is a call-site $dijkstra().CS1$ and the bounds formed assign the call-site and the callee-function of size ($qcount().Size = 16$bytes) to page $P2$. The page $P2$ is chosen because of an already existing bound for function $dijkstra()$ in $PG_2.B2$. The bounds formed are given in Equation (6.12). Here, the call-site is at an offset of 360 bytes from $dijkstra().Pos$, and the bound $B2$ which includes both $tb1$ and $tb2$ assures page-switch reduction.

$$\begin{aligned}
tb1 &= 2048 \geq dijkstra().Pos + 688 \geq 1024 \\
tb2 &= 2048 \geq qcount().Pos + 16 \geq qcount().Pos > 1024 \\
B2 &\leftarrow tb1 \ \& \ tb2
\end{aligned} \tag{6.12}$$

The next element in the ELEMENTS_LIST is the loop $dijkstra().L2$ of size (472 bytes), and when analyzed by the function ASSIGN_BOUNDS_LP(), generates bounds over the nested loop structure $L2 - L3$ in the function $dijkstra()$ to be contained within the page $P2$, is added to array $PG_2.B2$ Equation (6.13). For the next element $dijkstra().CS4$, since the callee function $enqueue()$ was earlier assigned to PG_1, a conflict arises in the bounds formed and therefore no bound is formed for this element. The next subsequent call-site $djikstra().CS2$ generates the

corresponding bounds to place function $dequeue()$ within page PG_2 Equation (6.14).

$$\begin{aligned} &PG_2.B2 = 2048 \geq dijkstra().Pos + 648 + 472 \\ &PG_2.B2 \leftarrow PG_2.B2 > dijkstra().Pos + 648 > 1024 \end{aligned} \tag{6.13}$$

$$\begin{aligned} &tb1 = 2048 \geq dijkstra().Pos + 702 \geq 1024 \\ &tb2 = 2048 \geq dequeue().Pos + 184 \geq qcount().Pos > 1024 \\ &B2 \leftarrow tb1 \;\&\; tb2 \end{aligned} \tag{6.14}$$

Assigning Function Start Address ($FN.Pos$)

The set of inequalities within the array $PG_r.Bounds$ for each page is evaluated to derive the smallest

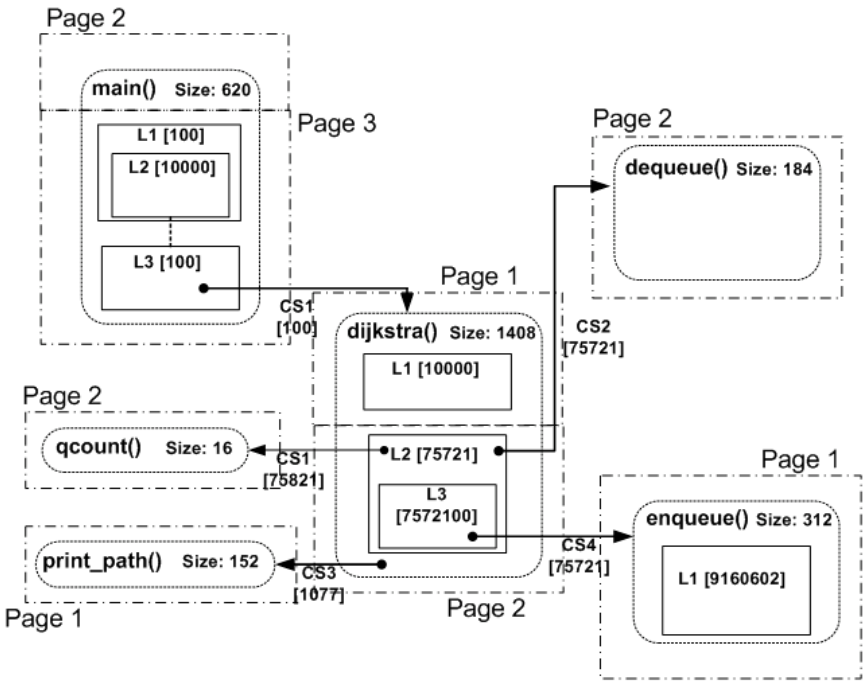


Figure 6.13: Optimized DCFG of *dijkstra* program with page demarcations.

integer value that can be assigned to the variable $FN_x.Pos$ for all functions having bounds within that page. Figure 6.13 gives the page allocations for the *dijkstra* benchmark after applying optimal placement of the functions by our B2P2 heuristic. The bounds formed in Equation (6.10), Equation (6.12), Equation (6.13) and Equation (6.14) are evaluated to achieve the required placement conditions for the functions. This optimal placement required 16 bytes of `nop` instructions to align function $dequeue()$ to a page-boundary. Our experiments demonstrate that this procedure placement achieves 52% reduced page-switch count with $< 2\%$ performance variation.

Heuristic Runtime

For a program with m procedures, n total loops, and c total call-sites, the size of the list ELEMENTS_LIST is $N = m + n + c$ governs the runtime. The list when implemented as a *heap* structure takes $O(log\ N)$ time for insertion and constant time for extraction. The **for** loop in Algorithm 3 runs for $O(N)$ iterations and within each iteration the functions to form bounds take constant time. The overall runtime of the B2P2 heuristic is thus bounded by $O(N) = O(m+n+c)$.

Experimental Setup

We have implemented our B2P2 heuristic as a profile based compiler post-pass optimization technique. For our experiments, we use the SimpleScalar [14] sim-outorder cycle-accurate simulator (implemented with the *Use-Last* i-TLB architecture), modified to count the total number of instruction-TLB page-switches for a program, configured to resemble the architectures of the Intel XScale [48] embedded processor with *tiny-page* (page-size = 1KB) configuration.

In our experiments to demonstrate the application of our B2P2 heuristic over a wide variety of uni-processor embedded systems, we have isolated benchmark programs (from the *MiBench* [36] benchmark suite) that represent different code varieties. The program in assembly language(*.s* format), is first compiled using the *GCC* (version 2.7.2.3) $-O2$ option. It is then profiled through execution in our processor simulator to populate the tuple hierarchy as discussed above. The output of our B2P2 heuristic is the values for function start addresses ($FN_x.Pos$). Padding using `nop` functions (each of size 8 bytes) are introduced to align the functions to page-boundaries.

Experiments

Benchmark from *MiBench*	**Page-Switches due to**						**PSR**	**Perf. Varn.**
	Function-Calls		**Loops**		**Sequential Access**			
	Orig	**Opt**	**Orig**	**Opt**	**Orig**	**Opt**		
Dijkstra	229517	2354	0	10000	86734	139791	52%	$< 1\%$
Patricia	23580	900	0	0	4336	456	95%	-3%
Bl_dec	94592	15592	311876	50	524453	284244	68%	-3%
Bl_enc	15592	15592	50	50	213671	213671	0%	0%
Sha	88	4885	623744	0	14623	4873	98%	$< 1\%$
a_caudio	688	688	0	0	685	685	0%	0%
a_daudio	688	688	1368866	0	1370	685	100%	-8%
fft	8196	8196	8202	0	2074	13	56%	$< 1\%$
fft_inv	16388	16388	16395	0	4123	13	56%	$< 1\%$

Table 6.1: Table showing page-switches due to function-calls, loops and sequential-executions, in benchmark applications

In this section we describe in detail our experimental results over an implementation of our B2P2 heuristic in the embedded processor simulator.

Overall Page-Switch Reduction

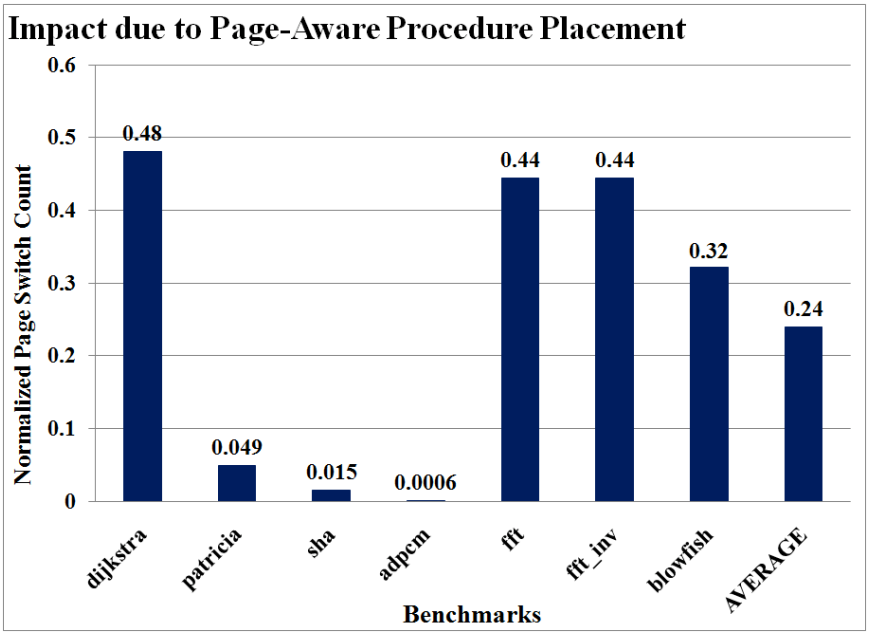


Figure 6.14: Impact of Code Placement on Page Switch Count.

In Figure 6.14, the *normalized page-switch count* of each benchmark application, normalized to the page-switch count of the baseline (un-optimized) program, is plotted. The average page-switch reduction achieved over the set of experimented benchmarks, is indicated by the bar on the right of the graph. Applications *patricia* and *adpcm* are characterized by procedures of large size and a number of call-sites with high call-counts. These characteristics of the program are identified by the B2P2 heuristic through the use of sorted ELEMENTS_LIST structure, and were therefore optimally placed to achieve significant page-switch reduction. On the other hand, the *fft* benchmark is dominated by the three nested loop structures in the *fft_float()* function. Owing to the sizes of these loops and the containing function, the B2P2 heuristic greedily binds the nested loops to a page, eliminating the dominating page-switches but causing smaller interfering function-calls and function sequential-executions to cause page-switches.

Over the set of benchmarks experimented, we observe an average of 76% reduction in the page-switch count, with less than 2% variation in performance and maximum variation of -8% (in *adpcm*). The achieved active power savings through our B2P2 heuristic (directly proportional to the page-switch reduction), is over and above that achieved through the *Use-Last* TLB architecture alone.

Program Page-Switches: break-up

In Table 6.1, the page-switches due to each instruction type (loops, function-calls or function blocks) in the program is tabulated for the set of benchmark applications used in our experiments. This tabulation portrays the impact of our greedy heuristic on the program page-switch count. The last two columns describe the overall page-switch reduction and performance variation of the optimized program when compared with the original program. A performance variation > 0 indicates runtime decrease while < 0 indicates runtime increase after optimization. For example, in *dijkstra* the original program placement did not cause any page-switches due to loops, but in optimizing for reduced total page-switch count, an overall reduction of 52% (with ¡ 1% variation in performance) was achieved through code placement, which resulted in a loop (iteration count = 10000) to span across a page-boundary causing increase in page-switches due to loops.

Our code placement technique aims to achieve maximum possible page-switch reduction on a given program and the benchmark *Blowfish* demonstrates such a condition. The *encode* and *decode* functionality of this benchmark, use the same code but for an if condition that chooses one loop over the other in the *bf_cfb64_encrypt()* function. Each of these loops have an iteration count 311825 equal to 33% of the program's total page-switch count. The page-switch count of the original program, incurred minimum number of total page-switches for the *encode* functionality, but owing to the use of different loops in the *bf_cfb64_encrypt()* function, procedure placement optimization was possible on the *decode* functionality resulting in an average of 68% page-switch reduction.

6.4 Summary

Most modern processors implement virtually addressed physically tagged caches, where virtual to physical address translation (using TLB) is required on every cache access. The *Use-Last* TLB architecture proposed in [37] reduces the TLB power consumption, if the same page is accessed successively; which was effective for the instruction-TLB. However, the approach was ineffective for data-TLB, because data accesses do not exhibit high locality as compared to instructions. Through analysis of the software application, system information could be extracted to develop smart compiler-level optimizations that efficiently modify the data access pattern such that the *Use-Last* d-TLB architecture shows reduced power consumption. In addition, we observe that by optimizing the placement of code within a binary, the power consumption of the instruction-TLB (equipped with *Use-Last*) can be further reduced.

Chapter 7

COMPILER-AWARE ARCHITECTURES

"Recently coarse-grained reconfigurable architectures (CGRAs) have drawn increasing attention due to their power-efficiency and flexibility. While many CGRAs have demonstrated impressive performance improvements at low energy cost, the effectiveness of CGRA platforms ultimately hinges on the compiler. Existing CGRA compilers do not model the details of the CGRA architecture, due to which they are, i) unable to map applications, even though a mapping exists, and ii) use too many PEs to map an application."

In this chapter, we model several CGRA details in our compiler and develop a graph mapping based approach (SPKM) for mapping applications onto CGRAs. On randomly generated graphs our technique can map on average 4.5X more applications than the previous approaches, while using fewer CGRA rows 62% times, without any penalty in mapping time. We observe similar results on a suite of benchmarks collected from Livermore Loops, Multimedia and DSPStone benchmarks.

7.1 Introduction

However, the success of CGRAs critically hinges on the efficient mapping of applications onto it so as to exploit the parallelism in the application and utilize minimum computation resources of the CGRA. Minimizing the number of computation resources in CGRAs is an extremely important goal, as it directly translates into either reduced power consumption, or increased throughput. The problem of mapping an application onto a CGRA to minimize the number of resources used has been shown to be NP-complete, and therefore several heuristics have been proposed. However existing heuristics do not consider the details of the CGRA architecture. In particular,

PE Interconnection Most existing CGRA compilers assume that the PEs in the CGRA are connected only in a simple 2-D mesh fashion, i.e., a PE is connected to it's neighbors only. However, in most existing CGRAs each PE is connected to more PEs than just the neighbor. In Morphosys, a PE is connected to , in RSPA, each PE is connected to .

Shared Resources Most CGRA compilers assume that all PEs are similar in the sense, that an operation can be mapped to any PE. However, modern CGRAs, in order to reduce the cost, power and complexity, do not include the multiplier in each CGRA. Few multipliers are made available as shared resources in each row. For example, RSPA has 2 shared multipliers in each row.

Routing PE Most CGRA compilers cannot use a PE just for routing. This implies that in a 4x4 CGRA, it is not possible to map application DAGs in which any node has more than 4 degree. However,

most CGRAs allow a PE to be used for routing only, which makes it possible to map any degree DAGs onto the CGRA.

Owing to the simplistic model of CGRA architecture in the compiler, existing CGRA compilers are i) unable to map an application on the CGRA, even though it is possible to map them. ii) uses too many PEs in the solution.

In this chapter,

- we formulate the application mapping problem onto CGRA considering the arbitrary PE interconnections, shared resource constraints, and routing PEs.
- we develop an ILP solution to solve the application mapping problem to minimize the number of rows used in the CGRA.
- we propose a graph drawing based approach to map applications onto CGRAs which can map 4.5X more randomly generated application DAGs than previous approach, while using less rows 62% of time, without any mapping time penalty.

7.2 Notation and Definitions

Since applications spend most of their time and energy in loops, in this paper, we focus on mapping loops to CGRAs. Significant power and performance trade-offs can be made by unrolling the loop. Therefore, the first step in mapping applications onto CGRAs is to first unroll the loops to meet the power and performance requirements. The focus in this paper is solving the problem of mapping the kernel of a given loop to a CGRA while minimizing the number of resources required.

Loop kernel

The loop kernel can be represented as a Directed Acyclic Graph (DAG), $K = (V, E)$, where the set of vertices V are the operations in the loop, and for any two vertices, $u, v \in V$, $e = (u, v) \in E$ iff the operation corresponding to v is data dependent on the operation u.

CGRA

An $M \times N$ CGRA can be represented as another directed graph $C = (P, L)$, where the elements of P, p_{ij}, where $1 \leq i \leq M$, and $1 \leq j \leq N$ are the PEs of the CGRA. For any two elements $p, q \in P$, $l = (p, q) \in L$ iff PE q can use the result that PE p computed in the previous cycle.

Application Mapping

An application mapping is a function $\phi : K \rightarrow C$, which in turn implies two functions, $\phi_V : V \rightarrow P$, and $\phi_E : E \rightarrow 2^L$. ϕ_V is an injective function that maps the operations to the PEs. This implies that each vertex $v \in V$ maps to a distinct PE $p \in P$, and that some PEs may be unused.

ϕ_E is a multi-valued function that maps data dependency edges $e \in E$ of the application kernel to a subset of interconnection links $ll \in 2^L$. Thus a dependence edge can be mapped onto a set of interconnection links on the CGRA, starting from $\phi_V(u)$, and ending at $\phi_V(v)$.

Path Existence Constraint

If $\phi_E(e = (u,v)) = ll \in 2^L$, then $\exists l_1 = (p_1, q_1) \in ll$ such that $p_1 = \phi_V(u)$, and $\exists l_2 = (p_2, q_2) \in ll$ such that $q_2 = \phi_V(v)$, and $\forall l = (p,q) \in ll$ such that $q \neq \phi_V(v)$ then $\exists l = (p', q') \in ll$, such that $p' = q$.

Simple Path Constraint

If $\phi_E(e = (u,v)) = ll \in 2^L$, then $\forall l_i, l_j \in ll$, if $l_i = (p,q)$, and $l_j = (r,s)$, then $q \neq s$. This implies that there are no loops in the path from $\phi_V(u)$ to $\phi_V(v)$, described by ll.

Routing Order

Under the path existence and the simple path constraint, a total order can be defined on the elements of ll. This total order, which we call routing order, and identified by $\prec$, identifies a unique path from $\phi_V(u)$ to $\phi_V(v)$. The total order is defined as:

1. $\forall l_i, l_j \in ll$, if $l_i = (p,q) \wedge \phi_V(u) = p$, then $l_i \prec l_j$,
2. $\forall l_i, l_j \in ll$, if $l_j = (p,q) \wedge \phi_V(v) = q$, then $l_i \prec l_j$
3. $\forall l_i, l_j \in ll$, if $l_i = (p,q) \wedge l_j = (q,r)$, then $l_i \prec l_j$

The routing order implies that the first element $\phi_V(u)$ is the smallest element, and $\phi_V(v)$ is the largest element. When two active interconnection links share a PE, the one that uses the PE as the source is larger than the one that uses it as a destination. If we arrange the PE vertices in increasing order, as defined by $\prec$, they describe the path from $\phi_V(u)$ to $\phi_V(v)$. The simple path constraint makes sure that if $ll \neq \phi$, then there exists a path from $\phi_V(u)$ to $\phi_V(v)$.

Routing PE

RPE for a data dependence edge $e \in E$ is the set of PEs, which are used to transfer data between two interconnection links. Thus, for any $e = (u,v) \in E$, if $ll = \phi_E(e)$, then $RPE^e = \{q | \forall l = (p,q) \in ll, q \neq \phi_V(v) \wedge p \neq \phi_V(u)\}$. We also define $RPE = \bigcup_{e \in E} RPE^e$.

Uniqueness of Routing PE

A routing PE can be used to route only one value.

No Computation on Routing PE

No computations can be performed on a routing PE.

Shared Resource Constraint

Most CGRAs have row-wise constraints, that arise from the fact that the rows share expensive resources, e.g. multipliers, memory buses etc. For example, there can be only two multiply operations in each row. To specify such constraints, we define an attribute "type" to each vertex of the kernel graph K and the number of shared resources, type t, within a row in C as S_t. Thus, $\forall v^t \in \{v \in V | v.type = t\}$ and $p^t_{ij} = \phi_V(v^t)$, $\sum_j |p^t_{ij}| \leq S_t$.

Utilized CGRA Rows

We define UR as the set of CGRA rows that are utilized in mapping the application K onto the CGRA C. Thus $UR = \{P_i | \forall j, p_{ij} \in Range(\phi_V) \vee p_{ij} \in RPE\}$.

7.3 Problem Formulation

Although the previously considered objective for the problem has been to minimize the number of PEs used. However, in practice, owing to the severe restrictions of the shared resource constraints, utilizing less number of rows is the most useful objective function. Utilizing lesser number of rows directly translates into increased opportunities for novel power and performance optimization techniques. For example, to reduce the energy consumption, a whole row of PEs may be power gated. In addition it might be possible to cleanly execute another application on the remaining rows to improve throughput. Therefore, we formulate our problem as follows: *Given a kernel DAG $K = (V,E)$, and a CGRA $C = (P,L)$, find a mapping $\phi(K)$, with the objective of min$|UR|$ or min$|RPE|$, under i) path existence, ii) simple path, iii) uniqueness of routing PE, iv) No computation on routing PE, and v) shared resource constraints.*

7.4 Related Work

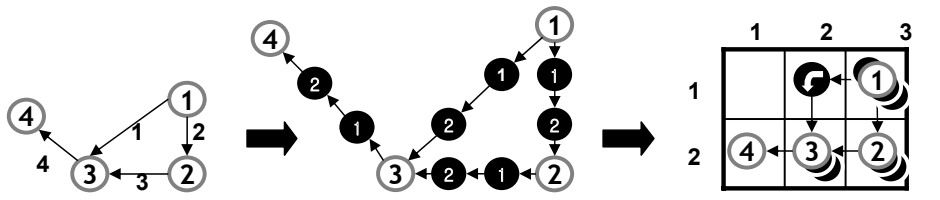
Various coarse grained architectures have been proposed as summarized in [40]. MorphoSys [95] consists of an 8×8 array of reconfigurable Cells coupled with a Tiny RISC processor. The Cell array performs 16-bits SIMD-style operations and each Cell has its own multiplier. RSPA [55] is based on MorphoSys, but it is not SIMD but MIMD-style. Furthermore, RSPA shares multiplier and pipelines it in order to reduce the chip size. REMARC [74] is 8×8 array of nano processors containing an ALU, data RAM, register file, and special purpose registers. ADRES [72] has an XML-based architecture description language to define the overall topology, operation set, resource allocation, timing, and an internal organization of each PE. Another CGRA is XPP [10]. It has 4×4 or 8×8 PE array and each PE has no multiplier.

The performance of CGRA critically hinges on the mapping algorithm. For MorphoSys, a compiler framework [101] to analyze SA-C programs, perform optimizations, and map the application was proposed. XPP has mapping algorithm described in [39]. However, their approach was evaluated only for simple loops. Another approach is DRESC [71] for ADRES architecture template. They exploit loop level parallelism by adopting modulo scheduling. This approach takes very long time for mapping and mapping results shows low utilization of PEs. In order to improve utilization, similar approach [81] using affinity graph was proposed.

However, all the previous application mapping approaches assume a very simple model of the CGRA, and do not model the complexities in the CGRA designs like row constraints, shared resources, limited load/store bandwidth, and irregular interconnections.

The work closest to ours is the Spatial mapping approach proposed in [2], in which the authors consider both shared multipliers and memory interface, and propose a spatial mapping technique. However, their approach often fails to find the mappings due to their simplistic model for the routing PEs. For example, their concept of channel PE is similar to the routing PE but it is added only for connecting two column-wisely unreachable PEs. Thus channel PE is not helpful for removing the diagonal edges or mapping node with more degree than the degree of PE. In addition, they can handle very restricted form of the applications. They only consider planar graphs and kernels which do not have loop-carried dependencies as their input applications. Since the input is limited to only simple binary trees and they do not use a PE only for a routing, their technique is not able to map non-planar DAGs onto CGRA. We compare our approach against [2] which is described above, and refer to their algorithm as AHN in this paper.

Application mapping onto a CGRA has been proven to be NP-complete [20], even in the special case when the application is a complete binary tree and the CGRA is a two dimensional grid with just the neighboring connections. Therefore, we formulate an Integer Linear Programming (ILP) solution for the problem.



$|V|=4, |E|=4, M=2, N=3, |R^e|=M*N - |V|=2,$
$v_{113}=1, v_{223}=1, v_{322}=1, v_{421}=1, \ldots$
$r_{1113}=1, r_{1222}=1, r_{2113}=1, r_{2213}=1, \ldots$
$q_{1113}=0, q_{1212}=1, q_{2113}=0, q_{2213}=0, q_{3123}=0, \ldots$

(d) Variables in ILP

Figure 7.1: Example of ILP formulation

When an edge e is mapped to a subset of interconnection links ll, such that $|ll| > 1$, it uses routing PEs. To accommodate the routing PEs, we add extra routing vertices on each edge of K. Since the exact number of routing PEs required for an edge $e \in E$ cannot be known until after the mapping is complete, we insert the maximum possible number of routing vertices. The upper bound on the number of routing PEs is $|P-V|$. Therefore we transform $K \rightarrow K' = (V', E')$ by inserting $|P-V|$ vertices on each edge $e \in E$. Figure 7.1 shows the transformation of adding the routing vertices (dark vertices) on every edge in K. Let R^e be the set of the inserted black vertices onto $e \in E$, and R as $R = \bigcup_{\forall e \in E} R^e$. The problem now translates into mapping K' to C. Unlike normal vertices $v \in V$, several black vertices, $r \in R$, can be mapped to the same PE $p \in P$ of the CGRA, including the PE to which a normal vertex $v \in V$ is mapped to. Now we define our ILP, on the transformed DAG K'.

Boolean Decision Variables

- v_{ikl} is 1 if i^{th} vertex $v_i \in V$ is mapped onto $p_{kl} \in P$.
- r_{ijkl} is 1 if j^{th} routing vertex $r_j \in R^{e_i}$ for $e_i \in E$ is mapped onto $p_{kl} \in P$.

- q_{ijkl} is 1 if j^{th} routing vertex $r_j \in R^{e_i}$ is mapped onto $p_{kl} \in P - Range(\phi_V)$, which means q corresponds to using an actual routing PE.

In Figure 7.1(d), r_{1113} is 1 because the first r for edge e_1 is placed on p_{13}, and q_{1212} becomes a routing PE because the second r for edge e_1 is placed on p_{12} where there are no operation v.

Objective Function

The objective function is to minimize the number of utilized rows. Thus,

$$min(\sum_{i}^{|V|}\sum_{k}^{M}\sum_{l}^{N} k \cdot v_{ikl} + \sum_{i}^{|E|}\sum_{j}^{|R^e|}\sum_{k}^{M}\sum_{l}^{N} k \cdot q_{ijkl}) \tag{7.1}$$

Constraints

$$\forall p_{kl} \in P, \sum_{i}^{|V|} v_{ikl} \leq 1 \tag{7.2}$$

$$\forall r \in R, \sum_{k}^{M}\sum_{l}^{N} r_{ijkl} = 1 \tag{7.3}$$

$$\forall p_{kl} \in P, \sum_{i}^{|V|} v_{ikl} + \sum_{i}^{|E|}\sum_{j}^{|R^e|} q_{ijkl} \leq 1 \tag{7.4}$$

$$\forall P_k, \sum_{i}^{|V|}\sum_{l}^{N} v^t_{ikl} \leq S_t \tag{7.5}$$

$$\forall e = (p, r_1) \in E'$$
$$\sum_{k_1}^{M}\sum_{l_1}^{N}\sum_{k_2}^{M}\sum_{l_2}^{N} Z_{k_1 l_1 k_2 l_2} \cdot v_{pk_1 l_1} \cdot r_{i1k_2 l_2} = 1 \tag{7.6}$$

$$\forall e(r_j, r_{(j+1)}) \in E', \;\; 1 \leq j \leq |R^e| - 1$$
$$\sum_{k_1}^{M}\sum_{l_1}^{N}\sum_{k_2}^{M}\sum_{l_2}^{N} Z_{k_1 l_1 k_2 l_2} \cdot r_{ijk_1 l_1} \cdot r_{i(j+1)k_2 l_2} = 1 \tag{7.7}$$

$$\forall e(r_{|R^e|}, q) \in E'$$
$$\sum_{k_1}^{M}\sum_{l_1}^{N}\sum_{k_2}^{M}\sum_{l_2}^{N} Z_{k_1 l_1 k_2 l_2} \cdot r_{i|R^e|k_1 l_1} \cdot v_{qk_2 l_2} = 1 \tag{7.8}$$

- Each $v \in V$ is mapped onto exactly one PE $p \in P$, Equation 7.2
- Each routing vertex, $r \in R$ is mapped onto exactly one PE $p \in P$, Equation 7.3

- Each $p_{kl} \in P$ can have only one operation, v or q, Equation 7.4
- Each $p_{kl} \in P$ can have at most S_t number of type t operations in the same row, Equation 7.5
- Equations 6-8 represents the interconnection links between $p_{kl} \in P$. We use a adjacent matrix table Z which contains all the information about the connections between p_{kl}. $Z_{k_1 l_1 k_2 l_2}$ is 1 if $p_{k_1 l_1}$ is directly connected to $p_{k_2 l_2}$ or if $k_1 = k_2$ and $l_1 = l_2$. Using Z, we represent all the possible connections when all $v, r \in V'$ are mapped onto $p_{kl} \in P$ using following three constraints.

Note that Equation 7.6- 7.8 are not linear. The contain products of boolean decision variables. Let a and b be boolean variables. The term $a \cdot b$ can be linearized by using an additional boolean variable t, and the following constraint : $t \geq a+b-1, \quad t \leq (a+b)/2$

7.6 Our Approach : Split-Push Kernel Mapping (SPKM)

Although the previously considered objective for the problem has been to minimize the number of PEs used. However, in practice, owing to the severe restrictions of the shared resource constraints, utilizing less number of rows is the most useful objective function. Utilizing lesser number of rows directly translates into increased opportunities for novel power and performance optimization techniques. For example, to reduce the energy consumption, a whole row of PEs may be power gated. In addition it might be possible to cleanly execute another application on the remaining rows to improve throughput.

Our approach of mapping a kernel onto a CGRA is based on the Split & Push algorithm [8] in graph drawing. Figure 7.2 shows an example of mapping 4-operation kernel K graph onto a mesh CGRA C.

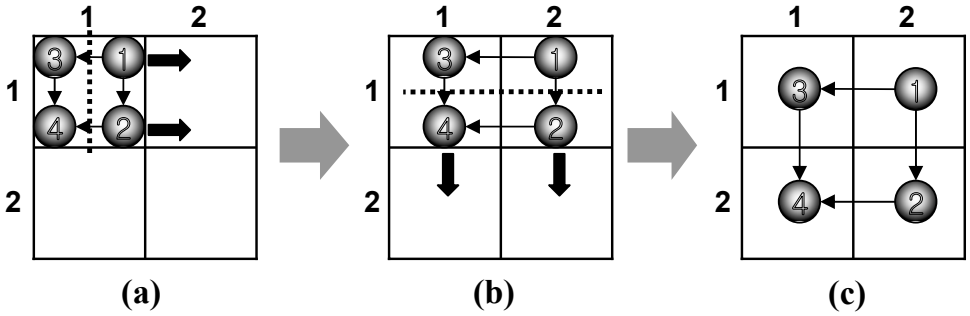


Figure 7.2: Split & Push Approach

The algorithm starts with a *degenerate drawing* where all the nodes in a graph are located at the same coordinate(1,1), as shown in Figure 7.2(a). In the first step, we locate each $v \in K$ using *cuts*. A cut is a plane orthogonal to one of the axis. It is shown by a dotted line in Figure 7.2. The cut *splits* all $v \in K$ into two groups. All v in one of the groups are *pushed* to new coordinate. Figure 7.2(b) shows the result

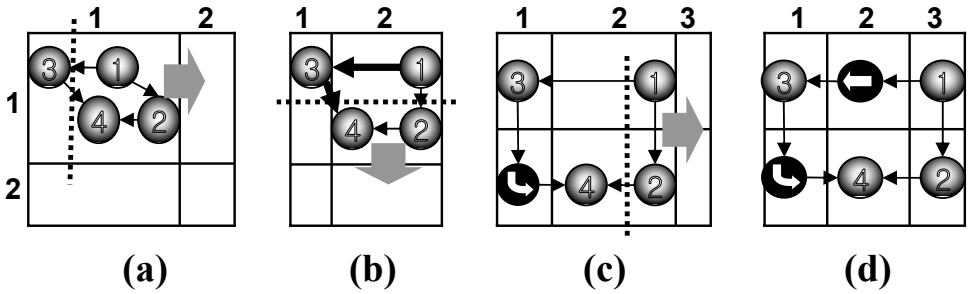


Figure 7.3: Formation of fork

of *split & push* along the dotted line. This split & push is repeated until every v has distinct coordinate like the drawing shown in Figure 7.2(c).

The crucial step in the split & push approach is finding a suitable cut. Consider the application of split & push applied to the same K in Figure 7.3. The final graph requires much more PEs than the solution in Figure 7.2. This is because v_3 is separated from other nodes in the first stage of split & push algorithm. This separation produces a *fork*. A fork is adjacent edges both cut by a split. Once there is a fork and the fork consists of n adjacent edges, $n-1$ bends (or dummy nodes, or routing vertices(PEs)) are required as $n-1$ edges in the fork become slant, which is not allowed in mesh graph drawing.

Forks can be avoided by finding a *matching-cut*. A matching-cut is defined as a set of edges which have no common node and whose removal makes the graph disconnected. The problem of finding a matching-cut in a graph is again an NP-complete problem [82].

In order to minimize the number of utilized rows in the mapping, we propose a three stage heuristic based on the split & push approach. Following three subsections explain *SPKM* by mapping the kernel graph $K = (V,E)$ shown in Figure 7.4(a), onto a 4×4 mesh CGRA $C = (P,L)$, shown in Figure 7.4(b).

We assume that in C, all $p \in P$ are connected to their first, as well as their second horizontal or vertical neighbors. Thus a PE is connected to at most 6 other PEs. We also assume that in a row, at most 2 load operations and one store operation can be scheduled. In K of Figure 7.4(a), we have ten operations including three loads (gray nodes) and one store (dark grey node).

Column-wise Scattering

This step distributes vertices v in V to minimum number of UR in the same column considering minimum number of forks and shared operations like multiply and load/store. First we compute the lower bound on $|UR|$ in the CGRA, as in $|UR|_{min} = max(\lceil |V|/|N| \rceil, \lceil L/L_r \rceil, \lceil S/S_r \rceil) = max(\lceil 10/4 \rceil, \lceil 3/2 \rceil, \lceil 1/1 \rceil) =$

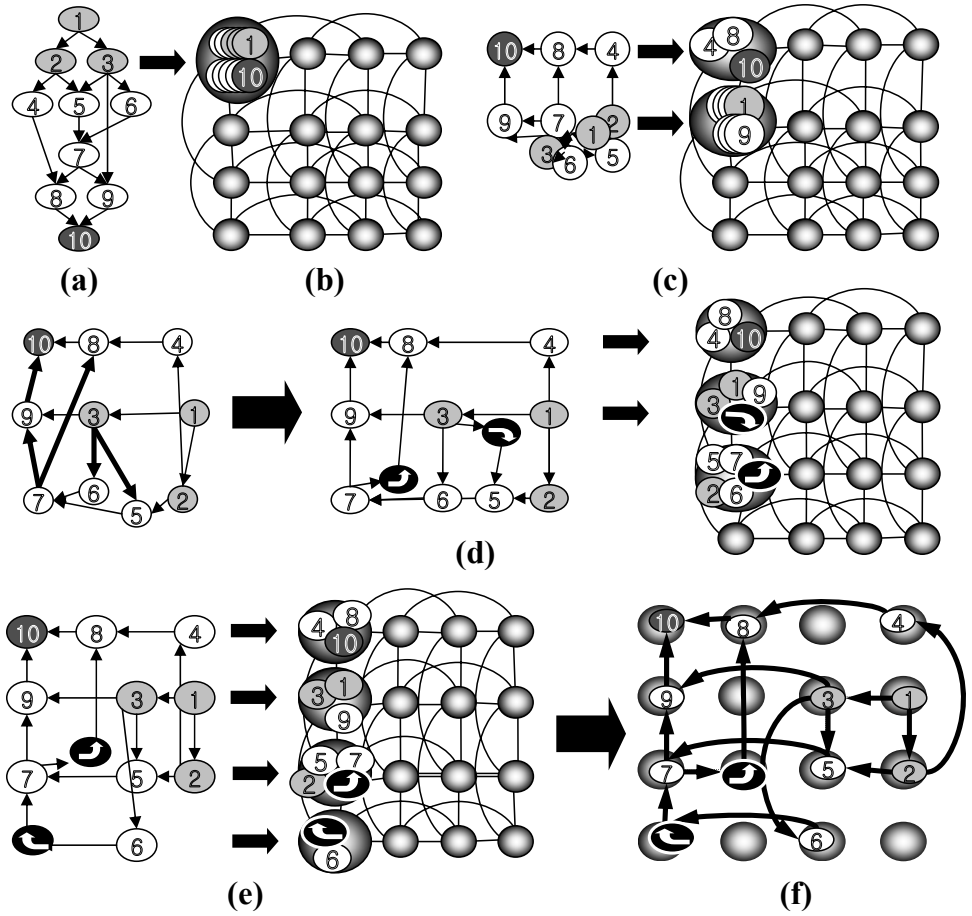


Figure 7.4: Mapping process example

3. We try to map K onto three rows of C. We distribute all v in K to p_{k1}, $1 \leq k \leq |UR|_{min}$. All v located at p_{k1} are separated into two sets and the nodes in one of the sets are pushed into $p_{(k+1)1}$. For example, all the nodes in p_{11} of Figure 7.4(a) are separated into two sets of nodes $\{v_4, v_8, v_{10}\}$ and $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$. The nodes in the set $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$ are pushed into p_{21} like in the Figure 7.4(c). Now the nodes $\{v_1, v_2, v_3, v_5, v_6, v_7, v_9\}$ are separated into two sets $\{v_1, v_3, v_9\}$ and $\{v_2, v_5, v_6, v_7\}$ again, and the nodes $\{v_2, v_5, v_6, v_7\}$ are pushed into p_{31}. The split & push is repeated until there are no *empty* p_{k1} in the $|UR|_{min}$. In each repetition, a matching cut is found to minimize the number of routing PEs.

ILP Solution of Matching Cut

Since matching cut problem is NP-complete, we solve it by formulating as an ILP. In the k^{th} repetition, the graph K^k consisting of all the nodes in p_{k1} are split into two disconnected graphs, and one of them, K^{k+1} is pushed into $p_{(k+1)1}$. K^1 is the same as K. We find a matching cut in K^k satisfying following ILP.

Objective Function

$$d = |\sum_{v \in V^k} v_{ik1} - \xi| \tag{7.9}$$

where v_{ik1} is 1 if the nodes are not pushed into $p_{(k+1)1}$, or 0 otherwise, and ξ is a constant restricting the number of nodes left in p_{k1}. As evenly distributing all nodes in K^k to all p_{k1} within $|UR|_{min}$ gives better chance for getting an optimal mapping, we set $\xi = \lfloor (N + |UR|_{min} - 1)/|UR|_{min} \rfloor$.

Constraints

- The first constraint restricts the number of nodes left in p_{k1} due to shared resources like memory buses or heavy computation resources. For example, the node v_1 in Figure 7.4(a) has one load primitive operation inside. So s_{111} is 1. In our CGRA, p_{kl} within one row share two read buses, S is 2.

- To minimize the forks, we have another constraint for all v_i^m with multiple edges in K^k.

$$\sum_{v_i \in V^k} v_{ik1}^t \leq S_t \tag{7.10}$$

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \leq \zeta_1 \ \ or$$
$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \geq 2 \cdot deg(v_{ik1^m}) - \zeta_2 \tag{7.11}$$

  where adj(v_i^m) is the set of nodes adjacent to v_i^m and $deg(v_i^m)$ is the degree of v_i^m. ζ_1 and ζ_2 are used for determining how many forks are allowed.

Equation 11 is not linear due to "or". In order to linearize this, we change this equation to Equation 7.12 and 7.13 using new constant η which is big enough.

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \leq \zeta_1, + \eta \cdot v_{ik1}^m \tag{7.12}$$

$$\sum_{v_j \in adj(v_i^m)} (v_{jk1} + v_{ik1}^m) \geq 2 \cdot deg(v_{ik1^m}) - \zeta_2 - \eta \cdot (1 - v_{ik1}^m) \tag{7.13}$$

When a *muti-degree-node* v_i^m, black node in Figure 7.5 and its adjacent nodes in K^k are determined to be pushed into $p_{(k+1)l}$, there are four ways to avoid generating forks. Figure 7.5(a) and (b) show

two possible cases where v_i^m is not pushed into $p_{(k+1)l}$ and $deg(v_i^m)$ or $deg(v_i^m)$-1 adjacent nodes are not pushed either. Figure 7.5(e) and (d) show the other cases where v_i^m is pushed into $p_{(k+1)l}$ and only 0 or 1 adjacent node is not pushed. Thus, if we want to allow no forks, we set $\zeta_1 = \zeta_2 = 1$. If there is no matching cut in the k^{th} repetition of split & push, we increase ζ_1 and ζ_2, allowing more forks.

The left kernel DAG K in Figure 7.4(d) shows the result of column-wise scattering. Because $|UR|_{min}$ is three, it attempts a mapping within three rows. Fortunately, it has a valid mapping in three rows within C in the end, instead it allows two forks.

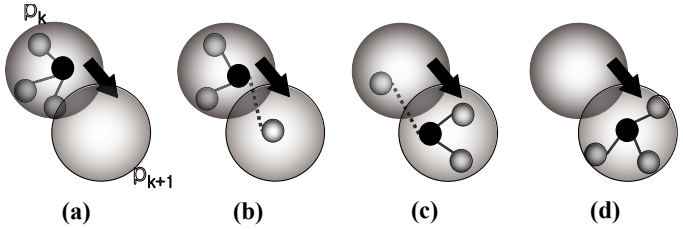


Figure 7.5: Fork minimization algorithm

Routing PE Insertion

After finishing column-wise scattering above, routing vertices should be inserted on each edge of each fork generated in the first stage of *SPKM* to route data via indirectly connected PE. In this step, we generate needed routing vertices and connect them with existing vertices. As (v_7, v_8) is an edge of the first fork in Figure 7.4(d), we need a routing PEs (black nodes) on it. A outing PE is also inserted into the edge (v_3, v_5) of the second fork. Sometimes there are no available PEs after routing PEs are inserted into K. For instance, the right K with routing PEs in Figure 7.4(d) has five nodes at p_{31} due to the insertion of a routing PE. Because we have four PEs in a row, this is an invalid mapping. In this case, we go back to column-wise scattering with increasing $|UR|_{min}$ by one. Figure 7.4(e) shows the result of column-wise scattering with $|UR|_{min}$ of four. We also need two routing PEs. One is on the edge (v_7, v_8) and the other is on (v_6, v_7). After the insertion of routing PEs, it is still a valid result.

Row-wise Scattering

In this last stage, we distribute all the nodes at p_{k1} to the nodes p_{kl} where $l \in [1, n]$. To avoid diagonal edge as well as edge crossing, all the nodes that have connections between different rows should be placed in the same column. For example, the nodes $\{v_3, v_5, v_6\}$ in Figure 7.4(d) are located in different rows but they have connections to each other. If the node v_6 is located in the fourth column while other two nodes v_3 and v_5 are located in the third column, we need a routing PE between v_3 and v_6.

7.7 Experiments

Experimental Setup

We test *SPKM* on a CGRA called RSPA [55]. RSPA is a 16 PEs in which each PE is connected to 4 neighboring PEs, and also the 4 next neighboring PEs (PE interconnection). In addition, it has 2 shared multipliers in each row (shared resource), each row can perform two loads and one store (also shared resource), and it allows PEs to be used for routing (routing PE). However, when a PE is used for routing, it cannot perform any operation.

For quantitative estimation of the effectiveness of SPKM, we have devised a random kernel DAG generator. Our DAG generator randomly generated 100 DAGs are for each value of node cardinality from 5 to 16 nodes (1200 in total). Each DAG is generated according to following steps. First the number of nodes is set. For each node, the operations possible in each PE of RSPA are randomly assigned. At least one load operation should be assigned to leaf vertices and one store operation to root vertices respectably. Finally edges are inserted, satisfying that each node should not have more than two incoming edges. In addition, we also compare SPKM and AHN on a collection of benchmarks from Livermore loops, MultiMedia and DSPStone. All experiments are done on Pentium4 3GHz machines with 1GB RAM. We use glpk4.8 for solving our ILP formulations.

SPKM can map more applications

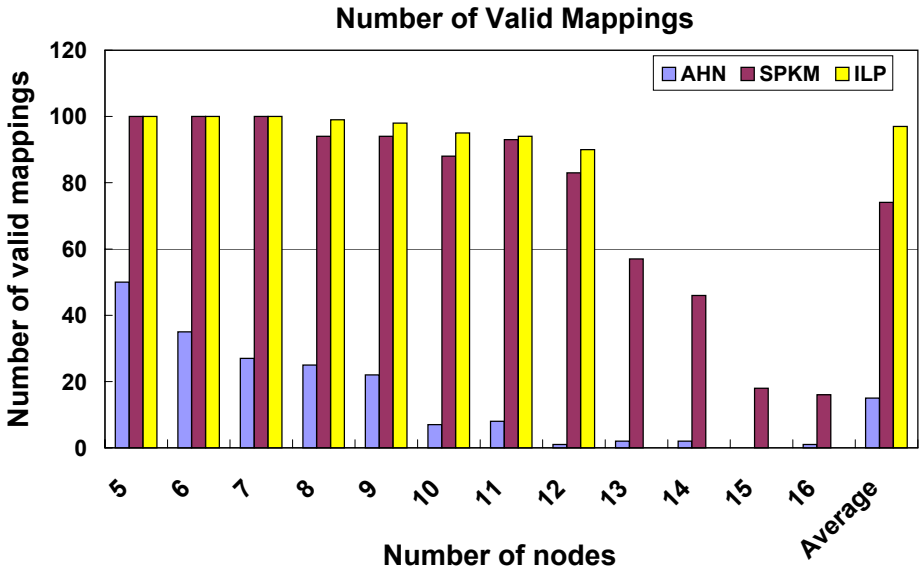


Figure 7.6: Number of applications mapped on CGRA validly

Figure 7.6 plots how many applications out of 100 applications can be mapped by the three techniques for each value of node cardinality. The X-axis represents the number of nodes that each input

application contains, and the Y-axis shows the number of valid mappings among 100 applications. ILP takes a lot of time for large input graphs. We stop the ILP solver after 24 hours. ILP cannot find a solution for graphs with 13 or more nodes, therefore, there are no ILP bars from 13 nodes in Figure 7.6. The main observation from this graph is that SPKM can on average map 4.5X more applications than AHN. It is interesting to note that the graph shows that the map-ability of SPKM over AHN increases with the increase in the number of nodes. This implies the effectiveness of our technique for the large applications.

SPKM can generate better mappings

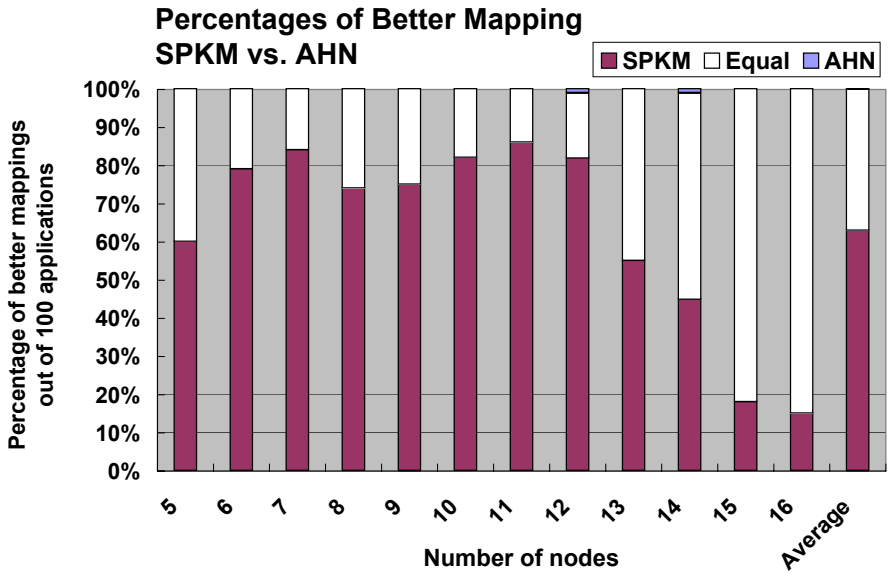


Figure 7.7: Percentage of better mapping for SPKM and AHN

In addition to being able to map more application than AHN, SPKM is also able to generate better mappings for the applications, in terms of the number of rows. Figure 7.7 plots how many applications out of 100, better, similar, or worse mapping is generated by SPKM and AHN. The white bars represent the number of applications in which SPKM and AHN maps with the same number of rows. For example, for the 100 applications which have 12 nodes, in 82 cases, SPKM can generate mappings which have fewer rows than AHN; AHN generated a mapping with fewer rows in 1 case, and in the rest 17 cases, SPKM and AHN generate mappings with similar number of rows. On average, SPKM can generate better mappings than AHN for 62% of the applications, and the similar or better mappings for 99% of the applications.

Figure 7.8 plots in how many applications out of 100 ILP generates better mapping than SPKM. Note that this graph has data only till 12 nodes since ILP cannot map large applications in reasonable

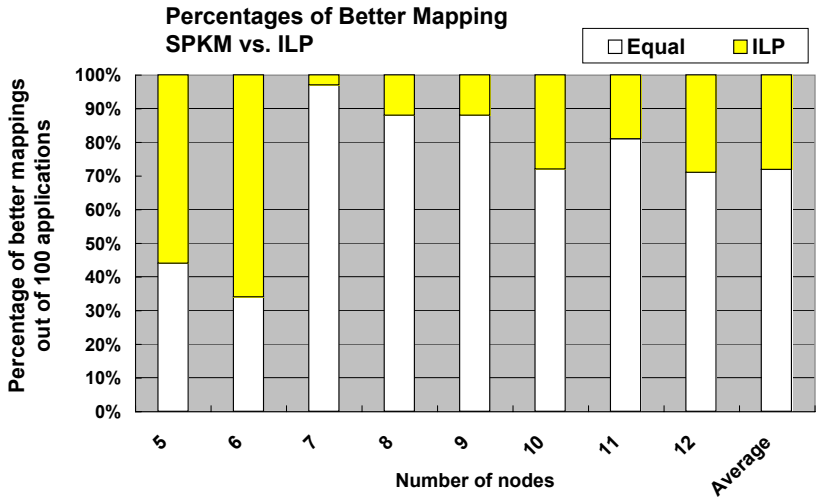


Figure 7.8: Percentage of better mapping for SPKM and ILP

time. Also note that there are only two kinds of bars. This is because SPKM can never generate better mapping than ILP. On an average, SPKM is able to generate optimal mapping in 72% of the applications.

SPKM has no significant mapping-time overhead

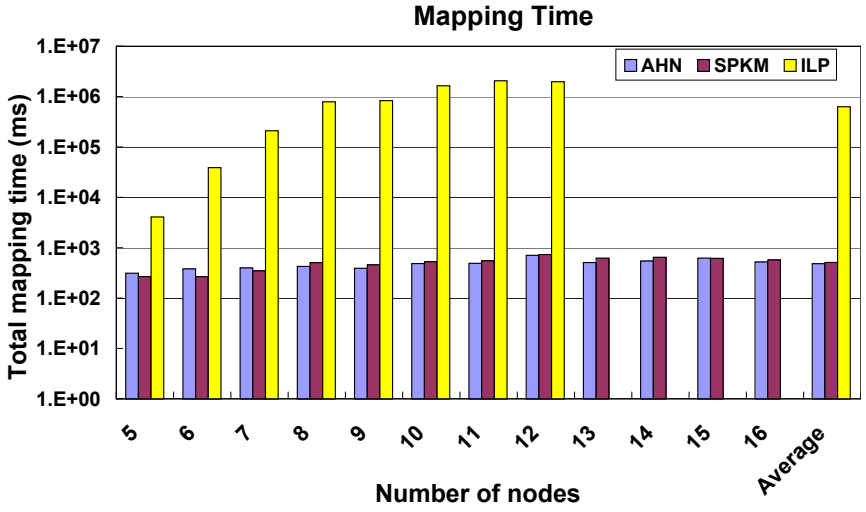


Figure 7.9: Total mapping time

SPKM generates effective mapping described above with minimal mapping time overhead. Figure 7.9 shows the average mapping time for all the three algorithms. Note that the Y-axis is a logarithmic scale. On average, SPKM has only 5% overhead in mapping time as compared to AHN, and both are much less than the time taken by ILP.

Real Benchmarks

To demonstrate the effectiveness and usefulness of SPKM, we compare the number of rows and the mapping time of SPKM and AHN for a set of benchmarks collected from Livermore loops, multimedia, and DSPStone. Since these applications are large, we try to map them onto a 6x4 RSPA.

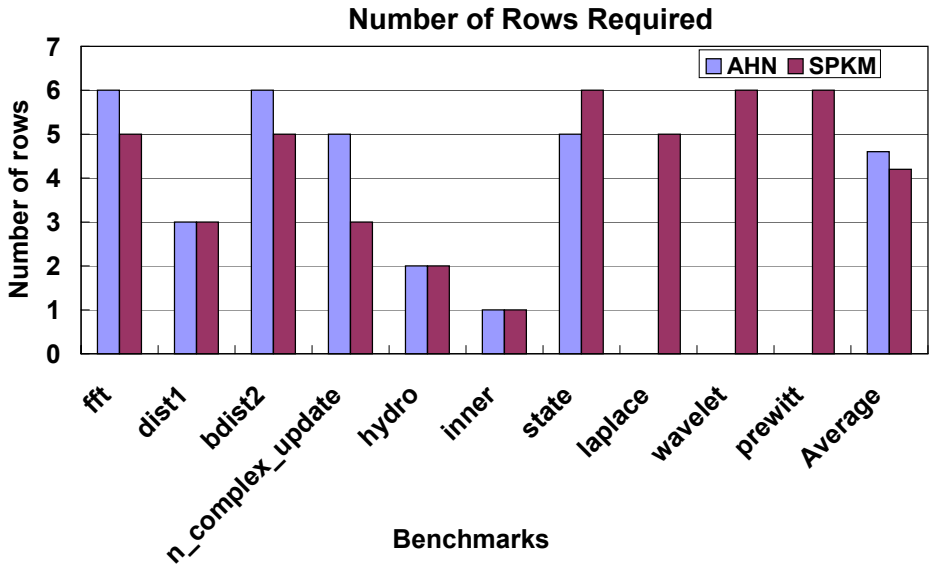


Figure 7.10: Number of rows for real benchmarks

Figure 7.10 shows the number of rows required for the mapping generated by SPKM and AHN. ILP is unable to find a mapping for any of these benchmarks in reasonable time. The first observation from this graph is that AHN is unable to map three of the applications, demonstrating that SPKM can map more applications than AHN. The second observation is that the mappings generated by SPKM uses less number of rows than AHN. SPKM uses fewer rows in 4 benchmarks out of 7, that can be mapped by AHN, demonstrating the goodness of mapping. For the benchmarks that AHN could map, SPKM uses just 2% more time than AHN.

7.8 Summary

While coarse-grained reconfigurable architectures (CGRAs) is emerging as attractive design platforms due to their efficiency as well as flexibility, efficient mapping of applications onto them still remains a challenge. Existing CGRA compilers assume a very simplistic architecture of the CGRA, due to which they are unable and ineffective. In this paper, we propose a graph drawing based based approach, *SPKM*, which takes in to account several architectural details of CGRA and is therefore able to effectively and efficiently map applications onto CGRAs. Our experiment demonstrate that *SPKM* can map 4.5X

more synthetic applications than previous approach, and generates better results in 62% of cases, with minimal (5%) mapping-time penalty. Results on benchmarks from Livermore loops, MultiMedia and DSPStone also convey the same. Our future work is to extend the *SPKM* approach to include dynamic reconfigurability.

[1] A. Agarwal, B. Paul, S. Mukhopadhyay, and K. Roy. Process variation in embedded memories: failure analysis and variation aware architecture. *Solid-State Circuits, IEEE Journal of*, 40(9):1804 – 1814, sep 2005.

[2] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 363–368, Leuven, Belgium, 2006.

[3] AMD Corp. AMD®Athlon Processor Product Data Sheet, 2007.

[4] E. ANDERSON. Lapack: Users' Guide, 1995.

[5] ARM. ARMv5 Architecture Reference Manual, 2007.

[6] D. F. Bacon, J.-H. Chow, D.-c. R. Ju, K. Muthukumar, and V. Sarkar. A Compiler Framework for Restructuring Data Declarations to enhance Cache and TLB Effectiveness. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 3. IBM Press, 1994.

[7] H. Balakrishnan and R. Garg. Multimedia Benchmarks: A Performance Comparison of Multimedia Programs on Different Architectures.

[8] G. D. Battista, M. Patrignani, and F. Vargiu. A Split&Push Approach to 3D Orthogonal Drawing. In *Proceedings of the 6th International Symposium on Graph Drawing*, GD '98, pages 87–101, London, UK, 1998. Springer-Verlag.

[9] R. Baumann, T. Hossain, S. Murata, and H. Kitagawa. Boron compounds as a dominant source of alpha particles in semiconductor devices. In *Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International*, pages 297 –302, apr 1995.

[10] J. Becker and M. Vorbach. Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC). In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, ISVLSI '03, pages 107–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Berkeley Design Technology Inc. The BDTI Benchmark suites.

[12] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *CASES '06: Proceedings of the 2006 international conference on Compilers,*

architecture and synthesis for embedded systems, pages 421–431, New York, NY, USA, 2006. ACM Press.

[13] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. *High-Performance Computer Architecture, International Symposium on*, 0:0171, 2001.

[14] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[15] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy. Cache size selection for performance, energy and reliability of time-constrained systems. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 923–928, Piscataway, NJ, USA, 2006. IEEE Press.

[16] E. Cannon, D. Reinhardt, M. Gordon, and P. Makowenskyj. SRAM SER in 90, 130 and 180 nm bulk and SOI technologies. *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 300–304, apr 2004.

[17] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler Optimizations for improving Data Locality. *SIGOPS Oper. Syst. Rev.*, 28(5):252–262, 1994.

[18] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[19] Y.-J. Chang. An ultra low-power TLB design. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1122–1127, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[20] J. Charles Oliver Shields. *Area efficient layouts of binary trees in grids*. PhD thesis, The University of Texas at Dallas, 2001. Supervisor-Ivan Hal Sudborough.

[21] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. *SIGPLAN Notices*, 36(5):286–297, 2001.

[22] G. Chen, M. Kandemir, M. J. Irwin, and G. Memik. Compiler-directed selective data protection against soft errors. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 713–716, New York, NY, USA, 2005. ACM Press.

[23] L. Chen and A. Avizienis. N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, page 113, Pasadena, CA, USA, jun 1995. IEEE Press.

[24] A. Chiyonobu and T. Sato. Energy-efficient Instruction Scheduling Utilizing CacheMiss information. *SIGARCH Comput. Archit. News*, 34:65–70, sep 2005.

[25] J.-H. Choi, J.-H. Lee, S.-W. Jeong, S.-D. Kim, and C. Weems. A Low Power TLB Structure for Embedded Systems. *Computer Architecture Letters*, 1(1):3 – 3, january-december 2002.

[26] L. T. Clark, B. Choi, and M. Wilkerson. Reducing translation lookaside buffer active power. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 10–13, New York, NY, USA, 2003. ACM.

[27] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. Irwin, A. Sivasubramaniam, and I. Kolcu. Compiler-directed Array Interleaving for reducing energy in Multi-Bank Memories. In *ASP-DAC '02*, pages 288–293, 2002.

[28] S. Devadas and S. Malik. A survey of optimization techniques targeting low power VLSI circuits. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 242–247, New York, NY, USA, 1995. ACM.

[29] B. Egger, J. Lee, and H. Shin. Dynamic Scratchpad Memory Management for Code in Portable Systems with an MMU. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–38, 2008.

[30] M. Ekman, P. Stenström, and F. Dahlgren. TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 243–246, New York, NY, USA, 2002. ACM.

[31] J. Gaisler. Evaluation of a 32-bit Microprocessor with Built-In Concurrent Error-Detection. *Fault-Tolerant Computing, International Symposium on*, 0:42, 1997.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[33] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 317–324, New York, NY, USA, 1997. ACM.

[34] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21:703–746, jul 1999.

[35] M. A. Gomaa and T. N. Vijaykumar. Opportunistic Transient-Fault Detection. *SIGARCH Comput. Archit. News*, 33(2):172–183, 2005.

[36] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[37] J. Haigh, M. Wilkerson, J. Miller, T. Beatty, S. Strazdus, and L. Clark. A Low-power 2.5-GHz 90-nm level 1 cache and memory management unit. *Solid-State Circuits, IEEE Journal of*, 40(5):1190 – 1199, may 2005.

[38] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[39] F. Hannig, H. Dutta, and J. Teich. Mapping of Regular Nested Loop Programs to Coarse-grained Reconfigurable Arrays - Constraints and Methodology. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 148, apr 2004.

[40] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[41] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[42] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 275 –278, 2001.

[43] X. Huang, B. T. Lewis, and K. S. McKinley. Dynamic Code Management: Improving whole program Code Locality in Managed Runtimes. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 133–143, New York, NY, USA, 2006. ACM.

[44] L. Hung, M. Goshima, and S. Sakai. Mitigating soft errors in highly associative cache with CAM-based tag. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 342–347, oct 2005.

[45] L. Hung, H. Irie, M. Goshima, and S. Sakai. Utilization of SECDED for Soft Error and Variation-Induced Defect Tolerance in Caches. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, apr 2007.

[46] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *Electron Devices, IEEE Transactions on*, 57(7):1527 –1538, jul 2010.

[47] Intel Corp. Intel Core 2 Extreme Mobile Processor.

[48] Intel Corp. Intel XScale®Core Developer's Manual, Dec 2003.

[49] Intel Corp. Intel®64 and IA-32 Developer's Manuals, May 2011.

[50] I. Issenin and N. Dutt. FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 808–813, Washington, DC, USA, 2005. IEEE Computer Society.

[51] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for Reducing dTLB Power. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 161 – 168, 2004.

[52] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Optimizing Instruction TLB Energy using Software and Hardware Techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 10:229–257, apr 2005.

[53] M. Kandemir, I. Kadayif, and G. Chen. Compiler-Directed Code Restructuring for Reducing Data TLB Energy. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 98–103, New York, NY, USA, 2004. ACM.

[54] S. Kayali. Reliability Considerations for Advanced Microelectronics. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, PRDC '00, page 99, Washington, DC, USA, 2000. IEEE Computer Society.

[55] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 12–17, Washington, DC, USA, 2005. IEEE Computer Society.

[56] J. Lee, B.-G. Nam, and H.-J. Yoo. Dynamic Voltage and Frequency Scaling (DVFS) scheme for multi-domains power management. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, pages 360 –363, nov 2007.

[57] J. Lee and A. Shrivastava. Static analysis to mitigate soft errors in register files. In *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE '09.*, pages 1367–1372, apr 2009.

[58] J.-H. Lee, G.-H. Park, S.-B. Park, and S.-D. Kim. A selective filter-bank TLB system. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 312–317, New York, NY, USA, 2003. ACM.

[59] K. Lee, A. Shrivastava, N. Dutt, and N. Venkatasubramanian. Partitioning techniques for partially protected caches in resource-constrained embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 15:30:1–30:30, oct 2010.

[60] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating Soft Error Failures for Multimedia Applications by Selective Data Protection. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 411–420, New York, NY, USA, 2006. ACM.

[61] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 17:1343–1347, September 2009.

[62] J.-F. Li and Y.-J. Huang. An error detection and correction scheme for RAMs with partial-write function. In *Memory Technology, Design, and Testing, 2005. MTDT 2005. 2005 IEEE International Workshop on*, pages 115 –120, Aug 2005.

[63] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Soft Error and Energy Consumption Interactions: A Data Cache Perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, pages 132–137, New York, NY, USA, 2004. ACM.

[64] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 340–349, Jun 1994.

[65] V. Loechner. PolyLib - A library of polyhedral functions, Feb 2010.

[66] A. Makhorin. GLPK (GNU Linear Programming Kit), 2000.

[67] S. Manne, A. Klauser, D. Grunwald, and F. Somenzi. Low power TLB Design for High Performance Microprocessors, 1997.

[68] M. Manoochehri, A. Ejlali, and S. G. Miremadi. Joint write policy and fault-tolerance mechanism selection for caches in dsm technologies: Energy-reliability trade-off. In *Proceedings of the 2009 10th International Symposium on Quality of Electronic Design*, pages 839–844, Washington, DC, USA, 2009. IEEE Computer Society.

[69] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2 – 9, Jan. 1979.

[70] F. McMahon. L. L. N. L. Fortran Kernels Test: MFLOPS, 1993.

[71] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 166 – 173, dec. 2002.

[72] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21224, Washington, DC, USA, 2004. IEEE Computer Society.

[73] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43 – 52, feb 2005.

[74] T. Miyamori and K. Olukotun. REMARC (abstract): REconfigurable Multimedia ARray Coprocessor. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, pages 261–, New York, NY, USA, 1998. ACM.

[75] S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. Measuring Architectural Vulnerability Factors. *IEEE Micro*, 23(6):70–75, 2003.

[76] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *PRDC '04: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society.

[77] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski. Critical Charge Characterization for Soft Error Rate Modeling in 90nm SRAM. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1879 –1882, may 2007.

[78] A. Nourivand, A. Al-Khalili, and Y. Savaria. Aggressive leakage reduction of SRAMs using error checking and correcting (ECC) techniques. In *Circuits and Systems, 2008. MWSCAS 2008. 51st Midwest Symposium on*, pages 426–429, aug 2008.

[79] N. Oh, S. Mitra, and E. McCluskey. ED4I: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180–199, Feb 2002.

[80] D. Parikh, K. Skadron, Y. Zhang, and M. Stan. Power-Aware Branch Prediction: Characterization and Design. *IEEE Transactions on Computers*, 53:168–186, 2004.

[81] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 136–146, New York, NY, USA, 2006. ACM.

[82] M. Patrignani and M. Pizzonia. The Complexity of the Matching-Cut Problem. In *Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '01, pages 284–295, London, UK, UK, 2001. Springer-Verlag.

[83] P. Petrov, D. Tracy, and A. Orailoglu. Energy-effcient physically tagged caches for embedded processors with virtual memory. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 17–22, New York, NY, USA, 2005. ACM.

[84] A. Peymandoust, T. Simunic, and G. De Micheli. Complex instruction and software library mapping for embedded software using symbolic algebra. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(8):964 – 975, aug 2003.

[85] R. Phelan. Addressing Soft Errors in ARM Core-based Designs. Technical report, ARM, 2003.

[86] D. K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[87] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM.

[88] W. Pugh. Counting solutions to Presburger formulas: how and why. *SIGPLAN Notices*, 29(6):121–134, 1994.

[89] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[90] J. Robertson. High dielectric constant gate oxides for metal oxide Si transistors. *Reports on Progress in Physics*, 69(2):327, 2006.

[91] L. R. Rockett Jr. Simulated SEU hardened scaled CMOS SRAM cell design using gated resistors. *Nuclear Science, IEEE Transactions on*, 39(5):1532–1541, Oct 1992.

[92] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. *Dependable Systems and Networks, International Conference on*, 0:389, 2002.

[93] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation Tables for Scheduling in the Presence of Incomplete Bypassing. In *CODES+ISSS '04: Proceedings of the international conference on Hardware/Software Codesign and System Synthesis*, pages 194–199, Washington, DC, USA, 2004. IEEE Computer Society.

[94] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 90–96, New York, NY, USA, 2005. ACM.

[95] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.*, 49(5):465–481, 2000.

[96] C. Slayman. Alpha Particle or Neutron SER-What Will Dominate in Future IC Technology, 2010.

[97] V. Sridharan, H. Asadi, M. B. Tahoori, and D. Kaeli. Reducing Data Cache Susceptibility to Soft Errors. *IEEE Transactions on Dependable and Secure Computing*, 3(4):353–364, 2006.

[98] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437 –445, dec. 1994.

[99] H. Tomiyama and H. Yasuura. Optimal Code Placement of Embedded Software for Instruction Caches. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 96, Washington, DC, USA, 1996. IEEE Computer Society.

[100] H. Tomiyama and H. Yasuura. Code Placement techniques for Cache Miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.

[101] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES '01*, pages 116–125, New York, NY, USA, 2001. ACM Press.

[102] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting Integer Points in Parametric Polytopes using Barvinok's Rational Function. *Algorithmica*, 48(1):37–66, 2007.

[103] T. Waho, K. Hattori, and K. Honda. Novel resonant-tunneling multiple-threshold logic circuit based on switching sequence detection. In *Multiple-Valued Logic, 2000. (ISMVL 2000) Proceedings. 30th IEEE International Symposium on*, pages 317 –322, 2000.

[104] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.

[105] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 203–209, New York, NY, USA, 2005. ACM.

[106] W. Zhang. Computing and Minimizing Cache Vulnerability to Transient Errors. *IEEE Des. Test*, 26:44–51, March 2009.

[107] X. Zhou and P. Petrov. Low-power cache organization through selective tag translation for embedded processors with virtual memory support. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, GLSVLSI '06, pages 398–403, New York, NY, USA, 2006. ACM.

[108] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, Dallas, 1994.

Appendix A

NEED FOR SOFT ERROR RESEARCH IN MOBILE SYSTEMS

A.1 Need for Soft Error Research

We computed the SER, of mobile and general purpose computers (on standard loads), using available SER estimates for outdated technology nodes. On extrapolating the SER values for modern processor caches, we arrive at a maximum of 4 probable failures per month. We believe that the soft error rate for modern and future mobile or general purpose processors, can be predicted to increase and is a cause for concern. The reasons for this conclusion are:

1. The SER extracted here donot include the impact of soft errors on sequential elements and combinational logic circuits in the processor.

2. With technology scaling the Q_{crit} of a gate reduces, thereby increasing the probability of a SEU and thereby translating to increased soft error failures.

3. Owing to the decrease in Q_{crit}, low energy neutron particles also cause soft errors in the system.

Processor (technology)	**Cache Size (Mbits)**	**Failures (in 10^9Hrs)**	**Failures (per week)**	**Failures (per year)**
General Purpose Processors				
Intel i7 (32nm)	64.5	103200	0.017	0.90
Intel i5 (32nm)	48.5	77600	0.013	0.68
Intel i3 (32nm)	24.5	39200	0.007	0.34
Mobile Processors				
Intel i7 (32nm)	64.5	103200	0.017	0.90
Intel i5 (32nm)	24.5	39200	0.007	0.34
Intel i3 (32nm)	24.5	39200	0.007	0.34
Apple A4 (45nm)	5.5	8800	0.001	0.08

Table A.1: Extrapolated SER of general purpose and embedded processors.

A.2 Evaluations

To estimate the soft error rate (SER) of a system, we model the SER in FIT/Mbit (1 FIT = 1 failure per 1 billion hours of execution). For our estimates, we use the SER value (at sea level) for the cache in the 90nm technology; which is equal to 1.6K FIT/Mbit[68]. We obtain the cache sizes of the processors from their corresponding specification documents, and extrapolate the SER of the processor caches at sea level in Table A.1. From the soft error rates thus extracted, we observe that with each technology generation, the number of cores and thus the size of cache in the processor increases along with an increase in the total number of sequential elements in the processor; which contribute to an increase in the overall soft error failure rate. Until recently, alpha-particles and high energy neutrons (of 100KeV$\sim$ 1GeV from

the cosmic background), have been known to cause soft errors in semiconductor devices [69]. With technology scaling, even low energy neutron particles (of 10meV$\sim$ 1eV) can cause soft errors in sub 45nm SRAM memory cells [96]. This is exacerbated by the fact that there are many more low-energy neutrons, than those with higher energies [46]. Therefore, we can conclude that the SER values observed here is likely to increase by atleast an order of magnitude, thus increasing the likelihood of attaining a FIT of one failure per-day.

Appendix B

POWER CONSUMPTION OF THE CPU IN A SMART PHONE

On an average, from the experimental results from [18], we observe that the CPU consumes around 12% $\sim$ 19% of the total power across common usage patterns. The proposed techniques in the thesis proposal aim to reduce TLB power, which consumes 17% of on-chip power in the Intel StrongARM processor. The proposed techniques achieve an overall power reduction (assuming the *use-last* TLB architecture is implemented) of 56% if i-TLB power and 18% of d-TLB power. From a global perspective, using the power consumption of the Intel StrongARM processor, the proposed techniques can achieve around 9.52% reduction in power when the i-TLB method is implemented, and can achieve around 3% when d-TLB techniques are implemented. Lacking availability of information that relates CPU power consumption and battery life, we can assume that proportion of CPU power reduced is directly proportional to the increase in battery life of an iPhone battery.

B.1 Experimental Deductions

Carroll et al. [18] in their work, present experimental results and thereby analyze the power consumption of a modern smartphone. In this, a breakdown of the power distribution to the various blocks in a smartphone is analyzed and validated (HTC Dream and Google Nexus One) over different usage patterns. The results of the analysis are as follows:

- Next to the LCD display, the CPU is the major power consuming block in general usage modes of the smartphone.
- In the suspended state (when the phone's LCD display is turned off and the phone is not actively used), the CPU consumes around 20% of the total power consumed. Since the CPU acts as the controller for enabling signals through the GSM module, and polling hardware interrupts to wake from the suspended state, the CPU performs tasks in the static mode. The static power consumption of the phone contributes largely to the overall power consumption of the smartphone.
- In the idle state (when the LCD display is ON, but the phone is not actively used), more applications or programs are executed on the processor. These tasks thus contribute to a power consumption of around 15%.
- During operations like audio playback and video playback, the control and execution of the required programs/data on the hardware audio and video accelerator blocks by the CPU. These tasks of the CPU contribute to an average of around 30% power consumption in these modes.
- Over a series of daily usage patterns, we observe that the CPU consumes around 12% $\sim$ 19% of the total power across the usage patterns. We observe that next to the graphics blocks and GSM module, the CPU is major contributor to the overall power consumption in the smartphone.

B.2 Observation

These measurements have been extracted over an older version of the smartphone, when no additional applications are executed on the system. When the number of applications used increases, the CPU becomes the only hardware block to manage, monitor, and schedule the applications on the smartphone. Thus, it is only evident that the overall power consumption proportion of the CPU will increase with increase in the functionality of the smartphone and an increase in the number of applications used. From our analysis of the results, and observing the current trend of smartphone applications and their use, we are justified in expecting the power consumption of the processor to be a significant contributor to the battery life of the smartphone. Therefore, we can conclude by saying that hardware, software or hybrid techniques developed to reduce the power consumption of mobile/embedded processors is valid and will help increasing the battery life and thus the usability of modern mobile systems.