B2P2: Bounds Based Procedure Placement for Instruction TLB Power Reduction in Embedded Systems

Reiley Jeyapaul   and   Aviral Shrivastava

Compiler-Microarchitecture Laboratory,
Arizona State University, Tempe, AZ - 85281, USA
{reiley.jeyapaul, aviral.shrivastava}@asu.edu

Abstract

High performance embedded processors are equipped with the Translation Look-aside Buffer (TLB) which forms the key ingredient to efficient and speedy virtual memory management. The TLB though small, is frequently accessed, and therefore not only consumes significant energy, but also is one of the important thermal hot-spots in the processor. Among the many circuit and microarchitectural techniques proposed to reduce TLB power consumption, the Use-Last *TLB is one very efficient technique in which power is consumed only when different pages are accessed in succession, i.e., when there is a* page-switch *[26]. Though the* Use-Last *technique is effective in reducing i-TLB power, there is scope to further improve its effectiveness by changing the relative code placement of the program. In this work, we formulate the code placement problem to minimize the page-switches in a program. We prove that this problem is NP-complete and propose an efficient* Bounds Based Procedure Placement *(B2P2) heuristic to efficiently reduce the program's page-switches. Our procedure placement technique delivers an average of* 76% *reduction in the instrucion-TLB power with negligible ($< 2\%$) impact on performance, over and above the reduction achieved by the* Use-Last *TLB architecture alone.*

Categories and Subject Descriptors C.1.0 [*Computer Systems Organization*]: Processor Architectures

General Terms Design, Measurement, Performance, Algorithms

Keywords Memory management, instruction TLB, Power, Code placement, Compiler technique, Embedded processors

1. Introduction

High-end embedded processors, like the the Intel XScale [17], MIPS S32 [24] support multi-tasking and virtual memory. The Translation Look-aside Buffer (TLB) is an important microarchitectural component of such embedded processors, and provides efficient virtual to physical address translation. Most embedded processors choose to implement a V/P cache configuration, in which the caches are indexed by virtual addresses, but have physical address tags. This is preferred to the P/P cache configuration (indexed by physical addresses and contain physical address tags), since TLB lookups *before* every cache access effectively utilize the cache latency for performance improvement. In comparison, the TLB lookup can be performed in parallel to the cache lookup in

V/P caches. Note that in V/P caches, TLB lookup is required even in the case of a cache hit to determine the page access permissions. On the other hand, in the V/V cache configuration (virtually addressed and virtually tagged), a TLB lookup is required only on a cache miss, and is therefore more power-efficient. The V/V caches are an interesting option, used in some high-performance general-purpose processors like DEC Alpha 21164 [5], but they suffer from synonym and address-mapping problems and are therefore not popular in embedded systems. The wide acceptance and popularity of the V/P cache configuration for most embedded applications is thus justified.

Another 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 [1] 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. [12] note that the TLBs can consume $20-25\%$ of the total $L1$ cache energy. Kadayif et al. [20] 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 [6, 8, 22, 23]. One effective microarchitectural technique for TLB power reduction, is the *Use-Last* TLB architecture [9, 26], 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 paper, we:

1. formulate the problem of reducing the total number of page-switches in a program, as a code placement problem. (Section 4, Section 5)

2. prove that the code placement problem for reduced page-switches is NP-complete. (Section 6)

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. (Section 7)

Our experiments on the Intel XScale microarchitecture [17], modeled on the SimpleScalar [3] cycle-accurate simulator, with a page size of 1KB, executing applications from the MiBench suite [14], demonstrate that our B2P2 heuristic can reduce the number of page switches by an average of 76%, with less than 2% performance variaton. Consequently, we expect 76% active power reduction over and above that achieved by the already effective *Use-Last* hardware technique.

2. Related Work

TLB power reduction is important not only to reduce the total energy consumed by the processor, but also to alleviate the high power density issue of the TLB in the processor. Several researchers have proposed efficient circuit-level, microarchitectural, software and hybrid (compiler and microarchitecture) techniques to reduce the power consumption of the TLB and the Memory Management Unit (MMU).

2.1 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. [23] 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 [22]. Through the use of selective filtering and banking mechanisms, the number of entries activated on each access is reduced and therefore efficient in embedded processors.

Choi et al. [8] 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 [6] 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. [19] 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.

2.2 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.

2.2.1 Software Techniques

Parikh et al. [25] 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. [7] 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.

2.2.2 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 [10]. 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. [21] 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 our recent work [18], 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.

2.2.3 Hybrid Approaches for instruction-TLB

One effective hybrid approach with the goal to reduce instruction-TLB power, is by Kadayif et al. [20], 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 [20], 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 [26], 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 [20]), while the key component of the *Use-Last* TLB is a latch (detailed discussion of the architecture is available in Section 3).

In our earlier work [18], 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.

3. The *Use-Last* TLB Architecture
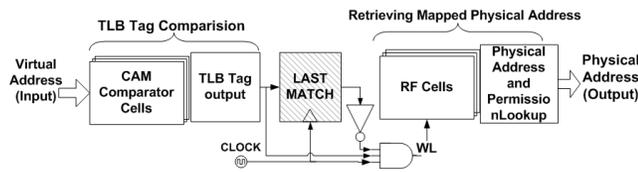
Figure 1. *Use-Last* TLB Architecture [26]: The *Use-Last* latch (shaded box) stores the previously translated page address. If the succeeding page accessed is the same, RF cells (for physical address and permission lookup) are not activated and stored data from the previous translation is bypassed to the output.

The *Use-Last* TLB architecture [26], described in Fig. 1 utilizes a modified TLB-CAM structure to reduce the per-access power consumption of the TLB. The virtual address input to the TLB is matched with the TLB tag through CAM structures, and then used to retrieve the mapped physical address from the lookup table (register file array). The lookup on the RF array is a power consuming process because of the bit-line and word-line drivers, and dynamic comparator circuitry involved in its operation. The key factor in this architecture design is the *Use-Last* latch used to store previously accessed TLB tag data. On a cache access, data in the *Use-Last* latch is compared with that from the current page. If found to be equal, the physical address and access permissions from the previous lookup are bypassed to the output, avoiding activation of the RF array cells. Otherwise, the *Use-Last* stores the current page address and performs lookup on the RF array to derive the physical address and permission information.

Switching energy is consumed by the *Use-Last* TLB structure, only when successive cache accesses are to different pages. The effectiveness of this technique was demonstrated on a 90-nm virtually addressed microprocessor cache memory subsystem functioning at 2.5 GHz with separate instruction and data cache structures of 32 KB each. The instruction-TLB demonstrated 75% power savings through circuit-level simulations using the *PowerMill* [15] simulator using a DSP benchmark [26]. Here, the total consumed power is distributed as 40% tag, 60% data array and less than 1% physical tag. The minimal tag power is due to the squashing of TLB lookups due to consecutive accesses to the same page.

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.

4. Page-Switch Reduction by code Placement

4.1 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.

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

4.2 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 atleast 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.

4.3 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.

4.3.1 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 in-

sertion 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.

4.3.2 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.

4.3.3 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 therfore 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 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%.

5. 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.

---

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

5.1 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.

5.2 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.

5.3 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 (1), where the individual components are represented by the equations: Equation (2)($PS_F(FN_x)$), Equation (3) ($PS_L(FN_x)$) and Equation (4) ($PS_S(FN_x)$).

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

(1)

5.3.1 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 Fig. 2, 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 (2).

$$PS_F = \sum_{\forall x: FN_x \in P} \sum_{\forall i: CS_i \in FN_x.CS[\,]} (FPS_F(FN_x.CS_i) + RPS_F(FN_x.CS_i)) \times CS_i.Count \qquad (2)$$

$$PS_L = \sum_{\forall x: FN_x \in P} \sum_{\forall i: LP_i \in FN_x.LP[\,]} (FPS_L(LP_i) + RPS_F(LP_i)) \times LP_i.Count \qquad (3)$$

$$PS_S = \sum_{\forall x: FN_x \in P} FN_x.Calls \times LonePB(FN_x) \qquad (4)$$
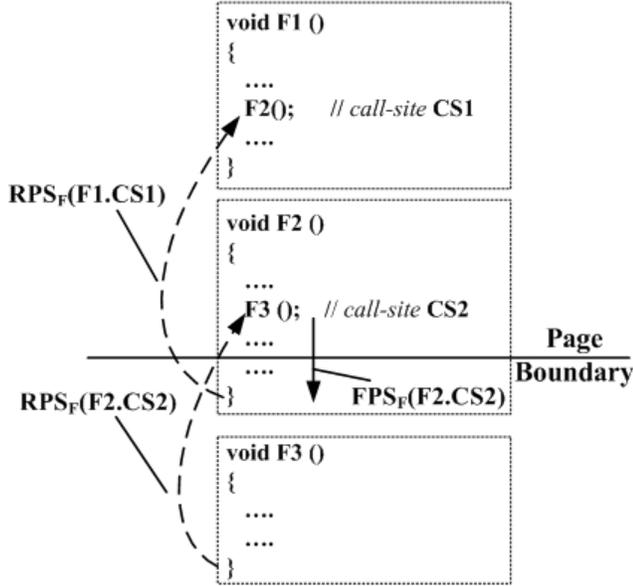
Figure 2. *Function-call Page-Switches*:
(i)$PS_F(F2)$ is the sum of the function's caller-to-callee page-switches ($FP_F(F2.C2)$) and callee-to-caller function-return page-switches($RPS_F(F2.C2)$).
(ii)$PS_F(F1)$ is the number of callee-to-caller function return page-switches($RPS_F(F1.C1)$).

5.3.2 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 Fig. 3(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 $Loop2$ of Fig. 3(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 (3).

5.3.3 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
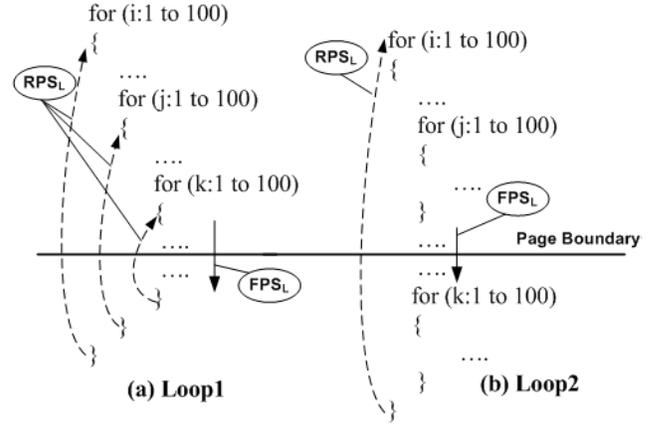
Figure 3. *Loop-execution Page-Switches* **for:**
(a)$PS_L(Loop1)$ is equal to the sum of its forward iteration $FPS_L = 100^3$ and loop-return $RPS_L = (100) + (100^2) + (100^3)$.
(b)$PS_L(Loop2)$ is the sum of $FPS_L = 100$ and $RPS_L = 100$.

within the function. If function $LonePB() : FN_x \to N$ 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 (4).

6. 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.

6.1 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).

6.2 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 $\mathbf{F}$ and edges $\mathbf{E}$, 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 $\sum_{c \in \mathbf{C'}} t(c) \leq \mathbf{M}$?

6.3 The Graph Partitioning Problem (GPP) [13]

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 $\sum_{e \in \mathbf{E'}} c(e) \leq \mathbf{J}$?

6.4 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.

6.5 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 [13]. 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.* ∎

6.6 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. [16] 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 [4, 27] 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 [2, 20] propose to enhance cache and TLB effectiveness through compiler-directed code transformation techniques. Bernhard et al. [11] 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 4) 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.

7. 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.

7.1 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 1 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.

Algorithm 1 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:   $type \leftarrow$ Get_Element_Type_Of (CP_i)
3:   **if** $type$ == LOOP **then**
4:     $LP_i \leftarrow addr(CP_x)$
5:     ASSIGN_BOUNDS_LP $(LP_i.FnId, LP_i)$
6:   **else if** $type$ == CALL-SITE **then**
7:     $CS_i \leftarrow *CP_x$
8:     ASSIGN_BOUNDS_CS $(CS_i.FnId, CS_i)$
9:   **end if**
10: **end for**

7.2 Illustration

7.2.1 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. Fig. 4 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 1, the program element with highest weight is taken, and its corresponding bounds assignment function (ASSIGN_BOUNDS_LP() or ASSIGN_BOUNDS_CS()) is executed.
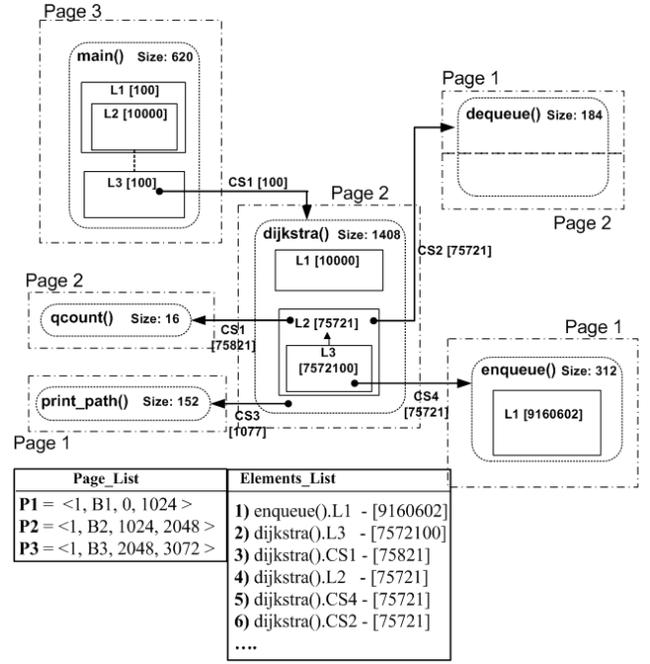
Figure 4. Original DCFG of *dijkstra* program with page demarcations.

7.2.2 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.

$$
\begin{aligned}
LP_i.SA &\leftarrow FN_x.Pos + FN_x.LP_i.Offset \\
LP_i.EA &\leftarrow FN_x.LP_i.SA + FN_x.LP_i.Size \\
bound &\leftarrow PG_r.EA \geq LP_i.EA > LP_i.SA \geq PG_r.SA
\end{aligned}
\tag{5}
$$

For the example in Fig. 4, 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). 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 (5). 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 (7) is formed. In Equation (6) and Equation (7), the $\leftarrow$ indicates that the newly formed bound is appended with the existing set of bounds.

$$
\begin{aligned}
PG_1.B1 &= 1024 \geq enqueue().Pos + 208 + 24 \\
PG_1.B1 &\leftarrow PG_1.B1 > enqueue().Pos + 208 > 0
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
PG_2.B2 &= 2048 \geq dijkstra().Pos + 1032 + 16 \\
PG_2.B2 &\leftarrow PG_2.B2 > dijkstra().Pos + 1032 > 1024
\end{aligned}
\tag{7}
$$

7.2.3 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.

$$CS.Addr \leftarrow FN_x.Pos + FN_x.CS_i.Offset$$
$$Callee.Pos \leftarrow (FN_x.CS_i.Callee).Pos$$
$$Callee.EA \leftarrow Callee.Pos + (FN_x.CS_i.Callee).Size$$
$$tb1 \leftarrow PG_i.EA \geq CS.Addr \geq PG_i.SA$$
$$tb2 \leftarrow PG_i.EA \geq Callee.EA \geq Callee.Pos \geq PG_i.SA$$
$$bound \leftarrow tb1 \ \& \ tb2$$
$$(8)$$

In Fig. 4, 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 (9). 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.

$$tb1 = 2048 \geq dijkstra().Pos + 688 \geq 1024$$
$$tb2 = 2048 \geq qcount().Pos + 16 \geq qcount().Pos > 1024$$
$$B2 \leftarrow tb1 \ \& \ tb2 \qquad (9)$$

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 (10). 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 (11).

$$PG_2.B2 = 2048 \geq dijkstra().Pos + 648 + 472$$
$$PG_2.B2 \leftarrow PG_2.B2 > dijkstra().Pos + 648 > 1024 \qquad (10)$$

$$tb1 = 2048 \geq dijkstra().Pos + 702 \geq 1024$$
$$tb2 = 2048 \geq dequeue().Pos + 184 \geq qcount().Pos > 1024$$
$$B2 \leftarrow tb1 \ \& \ tb2$$
$$(11)$$

7.2.4 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 integer value that can be assigned to the variable $FN_x.Pos$ for all functions having bounds within that page. Fig. 5 gives the page allocations for the *dijkstra* benchmark after applying optimal placement of the functions by our B2P2 heuristic. The bounds formed in Equation (7), Equation (9), Equation (10) and Equation (11) 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.

7.3 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.
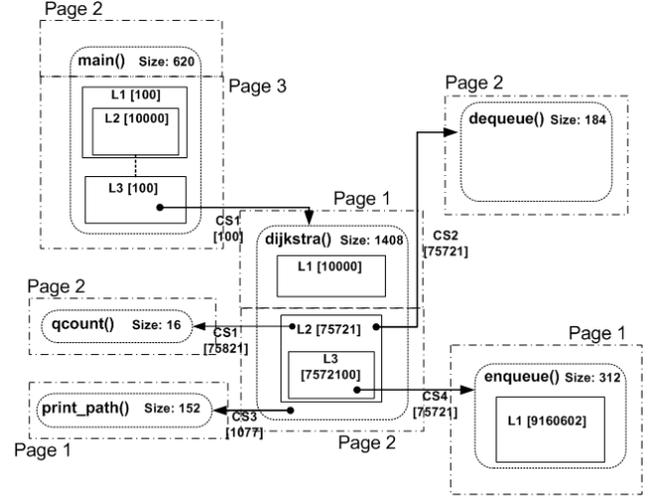
Figure 5. Optimized DCFG of *dijkstra* program with page demarcations.

The **for** loop in Algorithm 1 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 $O(N) = O(m + n + c)$.

8. Experimental Setup

We have implemented our B2P2 heuristic as a profile based compiler post-pass optimization technique. For our experiments, we use the SimpleScalar [3] 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 [17] 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* [14] 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 in Section 5. 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.

9. Experiments

In this section we describe in detail our experimental results over an implementation of our B2P2 heuristic in the embedded processor simulator (as described in Section 8).

9.1 Overall Page-Switch Reduction

In Fig. 6, 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

| Benchmark from *MiBench* suite | Page-Switches due to | | | | | | Overall Page-Switch Reduction | Overall Performance Variation |
| | Function-Calls | | Loops | | Sequential Accesses | | | |
	Original	Optimized	Original	Optimized	Original	Optimized		
Dijkstra	229517	2354	0	10000	86734	139791	52%	$< 1\%$
Patricia	23580	900	0	0	4336	456	95%	-3%
Blowfish_dec	94592	15592	311876	50	524453	284244	68%	-3%
Blowfish_enc	15592	15592	50	50	213671	213671	0%	0%
Sha	88	4885	623744	0	14623	4873	98%	$< 1\%$
adpcm_caudio	688	688	0	0	685	685	0%	0%
adpcm_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 1. Table showing page-switches due to function-calls, loops and sequential-executions, in benchmark applications
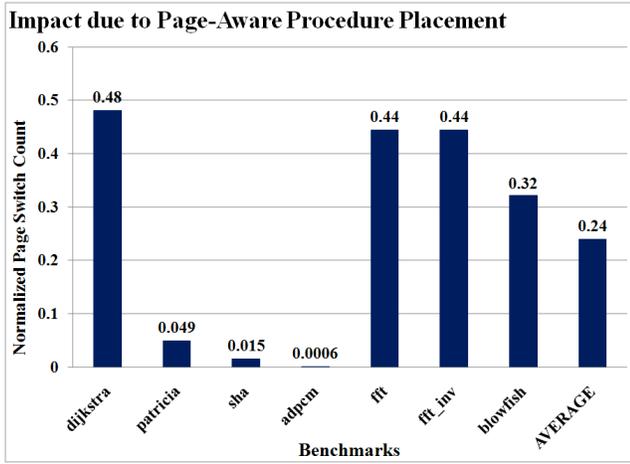
Figure 6. Impact of Code Placement on Page Switch Count.

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.

9.2 Program Page-Switches: break-up

In Table 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.

10. Summary

Most modern processors implement virtually addressed physically tagged caches, where virtual to physical address translation (using TLB) is required on every cache access. Coupled with the small page-sizes in embedded systems, the i-TLB contributes significantly to the overall power consumption of the processor. The *Use-Last* TLB architecture [26] is an efficient architectural modification that significantly reduces the i-TLB power for such systems. The *Use-Last* i-TLB reduces power consumption by performing a lookup only when the accessed page is different from the previous. Although *Use-Last* is an effective architectural technique, the power consumption can be further reduced by rearranging the code segments. To this end we formulate the procedure placement problem (PPP) to minimize the total number of page-switches in the program. We prove, through reduction from Graph Partitioning problem, that this code placement problem is NP-complete, and develop an ILP solution for a subset of the problem. We also propose an efficient *Bounds Based Procedure Placement* heuristic to reduce the program page-switches in the *linker-phase* of program compilation. Results over benchmarks from the *MiBench* suite show an average of 76% i-TLB power reduction over and above that by the *Use-Last* architecture alone, at negligible ($< 2\%$) performance variation.

11. Future Work

The code placement problem for minimized page-switches can also be approached at the basic-block granularity, taking advantage of the greater levels of freedom in movement of the code, to achieve page-switch reduction. Though such a finer granularity has an impact on code size and performance, an efficient technique could still be formulated to obtain significant page-switch reductions.On analyzing the optimized procedure placements in *dijktra* and *sha*, we observe that the B2P2 heuristic does not achieve maximum page-switch reduction, owing to the greedy nature of the heuristic. The PPP inherently lends itself to a network flow problem and therefore an efficient heuristic could be arrived at from that perspective. In [18], using the same *Use-Last* TLB architecture, we proposed code transformations to reduce the number of page-switches in the data TLB and thereby reduce d-TLB power. Evaluating the combined effects of these two techniques is an interesting research direction.

12. Acknowledgments

References

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

[2] 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.

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

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *SIGOPS Oper. Syst. Rev.*, 28(5):252–262, 1994.

[5] M. Cekleov and M. Dubois. Virtual-address caches part 1: Problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[6] 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.

[7] A. Chiyonobu and T. Sato. Energy-efficient instruction scheduling utilizing cache miss information. In *MEDEA '05: Proceedings of the 2005 workshop on MEmory performance*, pages 65–70, Washington, DC, USA, 2005. IEEE Computer Society.

[8] 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.

[9] L. T. Clark, B. Choi, and M. Wilkerson. Reducing translation lookaside buffer active power. In *ISLPED '03*, pages 10–13, New York, NY, USA, 2003. ACM Press.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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 *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[15] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski. The design and implementation of powermill. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 105–110, New York, NY, USA, 1995. ACM.

[16] 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.

[17] Intel Corporation. Intel XScale®Technology Overview, 2000.

[18] R. Jeyapaul, S. Marathe, and A. Shrivastava. Code transformations for tlb power reduction. In *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*, pages 413–418, Washington, DC, USA, 2009. IEEE Computer Society.

[19] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for reducing dtlb power. In *ISPASS '04*, pages 161–168, Washington, DC, USA, 2004. IEEE Computer Society.

[20] 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(2):229–257, 2005.

[21] M. Kandemir, I. Kadayif, and G. Chen. Compiler-directed code restructuring for reducing data tlb energy. In *CODES+ISSS '04*, pages 98–103, Washington, DC, USA, 2004. IEEE Computer Society.

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

[23] S. Manne, A. Klauser, D. Grunwald, and F. Somenzi. Low power tlb design for high performance microprocessors, 1997.

[24] MIPS Technologies. MIPS32®Architecture, 2003.

[25] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Instruction scheduling for low power. In *VLSI-SP '04*, pages 129–149, 2004.

[26] J. R.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. In *IEEE Journal of Solid State Circuits*, pages 1190–1199. IEEE Press, 2004.

[27] H. Tomiyama and H. Yasuura. Code placement techniques for cache miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.