Minimal Placement of Bank Selection Instructions for Partitioned Memory Architectures

BERNHARD SCHOLZ
The University of Sydney
and
BERND BURGSTALLER
Yonsei University
and
JINGLING XUE
University of New South Wales

We have devised an algorithm for minimal placement of bank selections in partitioned memory architectures. This algorithm is parameterizable for a chosen metric such as speed, space or energy. Bank switching is a technique that increases the code and data memory in microcontrollers without extending the address buses. Given a program in which variables have been assigned to data banks, we present a novel optimization technique that minimizes the overhead of bank switching through cost-effective placement of bank selection instructions. The placement is controlled by a number of different objectives, such as runtime, low power, small code size or a combination of these parameters. We have formulated the minimal placement of bank selection instructions as a discrete optimization problem that is mapped to a Partitioned Boolean Quadratic Programming (PBQP) problem. We implemented the optimization as part of a PIC Microchip backend and evaluated the approach for several optimization objectives. Our benchmark suite comprises programs from MiBench and DSPStone plus a microcontroller real-time kernel and drivers for microcontroller hardware devices. Our optimization achieved a reduction in program memory space of between 2.7% and 18.2%, and an overall improvement with respect to instruction cycles between 5.0% and 28.8%. Our optimization achieved the minimal solution for all benchmark programs. We investigated the scalability of our approach towards the requirements of future generations of microcontrollers. This study was conducted as a worst-case analysis on the entire MiBench suite. Our results show that our optimization (1) scales well to larger numbers of memory banks, (2) scales well to the larger problem sizes that will become feasible with future microcontrollers, and (3) achieves minimal placement for more than 72% of all functions from MiBench.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*

This project has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" (DP 0560190) and the University of Sydney R&D Grants Scheme "Speculative Partial Redundancy Elimination" (L2849 U3229).
Authors' addresses: Bernhard Scholz, School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia; email: scholz@it.usyd.edu.au; Bernd Burgstaller, Department of Computer Science, Yonsei University, Seoul, Korea; Jingling Xue, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia; email: jingling@cse.unsw.edu.au.

General Terms: Algorithms, Languages, Performance
Additional Key Words and Phrases: Partitioned Boolean Quadratic Programming, Bank Selection, Partitioned Memory Architectures

1. INTRODUCTION

Embedded systems have become an integral part of the infrastructure of today's technological society. They are prevalent in an ever-increasing range of applications, including consumer electronics, home appliances, instrumentation/measurement, automotive, communications and industrial control. Microcontrollers constitute the core of all embedded systems designs. The Semiconductor Industry Association's November 2005 forecast predicted the market for 4-, 8-, 16-, and 32-bit microcontrollers to grow to $12.8 billion in 2006. The reported share of 8-bit microcontrollers is 42%. Gartner Dataquest reports that the 8-bit market reached $5.5 billion in 2004 [Gartner Dataquest 2005].

The widespread use of 8-bit microcontrollers can be attributed to the following: (1) many embedded systems designs do not need the more costly, energy-burning and complex 16- or 32bit CPUs, (2) many embedded systems designs distribute small numbers of low-cost electronics instead of using one powerful and expensive core CPU, (3) embedded systems designs often employ 8-bit microcontrollers as low-cost subsystems of complex 32-bit hardware designs, and (4) there is a trend to add entry-level electronics intelligence to mechanics-based systems.

Bank switching is a common technique used for 8-bit microcontrollers which increases the size of code and data memory without extending the address buses of the CPU. The address space is partitioned into *memory banks*, and the CPU can only access one bank at a time. This bank is called the *active bank*. To keep track of the active bank the CPU's *bank register* stores the address of the active bank. A *bank selection instruction* is issued to switch between banks. Smaller address buses result in smaller chip die-sizes, higher clock frequencies and less power consumption. As an example, Motorola 68HC11 8-bit microcontrollers address a maximum of 64KB memory using their 16-bit address registers. This scheme allows multiple 64KB banks to be accessed although only one can be active at a time. As another example, the memory of the PIC 16F877A microcontroller is partitioned into four banks of 128B each. Other processor families have similar features such as Zilog's Z80 and Intel's 8051 processor families. Architectures such as Ubicom's 8-bit SX microcontroller organize their registers in register banks to shorten the cycle time, avoiding multi-porting [Nystrom and Eichenberger 1998; Ravindran et al. 2005; Kiyohara et al. 1993]. Bank-switched SRAMs are employed with ultra-low power sensors to achieve high code-density [Nazhandali et al. 2005] and allow the gating of individual memory banks [Hempstead et al. 2005; Hempstead et al. 2006].

The disadvantage of bank-switched architectures is the code-size and runtime overhead caused by bank selection instructions. Currently compilers provide limited support to generate bank-switched code. For example, GNU GCC for Motorola's 68HC11 and 68HC12 will compile a function declared with the `far` attribute by using a calling convention that takes care of switching banks when entering and

leaving a function. However, the GCC compiler does not eliminate redundant bank selection instructions. The CC5X compiler for mid-range PICmicro devices (from B Knudsen Data) expects the programmer to allocate variables to banks but will insert bank selection instructions automatically with no guarantee of optimal placement of the bank selection instructions. The PICC-18 for the PIC18Fxxx family appears to have automated both tasks under certain language restrictions. As far as the authors are aware, the bank switching schemes of existing compilers are ad hoc, and it remains a challenging research problem to generate efficient memory accesses for bank-switched architectures.

This work develops a compiler optimization for minimal placement of bank selection instructions in a bank-switched architecture. This problem is important because poor placement of bank selection instructions increases runtime, code-size, and power consumption. Given a program in which all variables have been assigned to banks (by the programmer or compiler), we present an optimization that inserts the minimum number of bank selection instructions in the program to guarantee the correct access to memory. The placement is controlled by a number of objectives such as runtime, low power, small code size or a combination of these parameters. We are only aware of an ad-hoc approach in this area [Kiyohara et al. 1993]. As we will explain in Section 3, the problem cannot be solved as a speculative partial redundancy elimination problem [Scholz et al. 2004; Knoop et al. 1994]. It can be seen as an extended code motion. However, bank selections impose dependencies which cannot be handled with classical code motion.

Most previous efforts on partitioned memory architectures focus on maximizing parallel data accesses to make memory banks simultaneously active [Cho et al. 2004; Leupers and Kotte 2001; Panda et al. 2001; Panda et al. 2000; Saghir et al. 1996; Sudarsanam and Malik 1995; Zhuang et al. 2002; Zhuge et al. 2002]. By enabling parallel memory accesses in a single instruction, one can increase memory bandwidth and thus improve program performance. Such partitioned memory banks are found in Motorola's DSP56000, Analog Devices' ADSP2016x and NEC's μPD77016. Some researchers re-organize the order of instructions and the layout of data, e.g., by loop transformations [Delaluz et al. 2000], to reduce energy consumption in partitioned memory architectures. In the case of heterogeneous memory banks such as scratchpad SRAM, internal DRAM and external DRAM, we refer to [Banakar et al. 2002; Li et al. 2005; Udayakumaran and Barua 2003; Verma et al. 2004] and the references therein for a number of compiler techniques proposed to perform automatic scratchpad management.

The contributions of this paper are as follows:

—We present a novel algorithmic approach to minimize the number of bank selection instructions in a partitioned memory architecture for a given cost metric.

—We formulate the problem as a form of Partitioned Boolean Quadratic Programming (PBQP). We present experimental evidence that PBQP is very efficient for real-world applications.

—We introduce an intra- and inter-procedural approach for placing bank selection instructions.

—We present our worst-case feasibility study using MiBench to show that our problem formulation can be solved almost optimally in polynomial time.

—We present our experimental results over a benchmark suite to show that our optimization can accommodate a variety of optimization objectives such as speed, space or a combination of both. We have implemented the optimization as part of a backend for a Microchip microcontroller. Microchip is the No. 1 8-bit microcontroller manufacturer with 45000 customers worldwide [Gartner Dataquest 2004].

The paper is organized as follows. In Section 2, we describe the background. In Section 3, we define and motivate the problem of minimizing the costs of bank selection instructions across basic block boundaries. The optimization algorithm is presented in Section 4. In Section 5, we present and discuss our experimental results. We draw our conclusions in Section 6.

2. BACKGROUND

A *basic block* [Muchnick 1997] is a sequence of statements in which flow of control can only enter from its beginning and leave at its end. A *control flow-graph*(CFG) is a directed graph $G = \langle V, E, s, e\rangle$ where V is the set of vertices representing basic blocks and E is the set of edges $E \subseteq V \times V$. Vertex s is the entry node (aka. *start node*) of the CFG and e is the exit node (aka. *end node*). The set of predecessors $preds(u)$ is defined as $\{w|(w,u) \in E\}$ and the set of successors $succs(u)$ as $\{v|(u,v) \in E\}$. For an edge $(u,v) \in E$, vertex u is the source and vertex v is the tail of the edge. A critical edge is an edge (u,v) for which $|succs(u)| > 1$ and $|preds(v)| > 1$, i.e. the source has several outgoing edges and the tail has several incoming edges[1]. A path π is a sequence of vertices $\langle v_1, \ldots, v_k\rangle$ such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. In a CFG, all vertices are reachable, i.e., there is a path from s to every other vertex in V.

The PBQP problem [Scholz and Eckstein 2002; Eckstein 2003] is a specialized quadratic assignment problem and is NP-complete. Consider a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$. A solution of PBQP is a simple function $h : X \rightarrow D$ where D is $\mathbb{D}_1 \cup \ldots \cup \mathbb{D}_n$; for each variable x_i we choose an element d_i in $\mathbb{D}_i$. The quality of a solution is based on the contribution of two sets of terms:

(1) for assigning variable x_i to the element d_i in $\mathbb{D}_i$. The quality of the assignment is measured by a *local cost function* $c(x_i, d_i)$.
(2) for assigning two related variables x_i and x_j to the elements d_i in $\mathbb{D}_i$ and d_j in $\mathbb{D}_j$. We measure the quality of the assignment with a *related cost function* $C(x_i, x_j, d_i, d_j)$.

Thus, the total cost of a solution h is given as

$$f = \sum_{1 \leq i \leq n} c(x_i, h(x_i)) + \sum_{1 \leq i < j \leq n} C\left(x_i, x_j, h(x_i), h(x_j)\right). \quad (1)$$

The PBQP problem seeks for an assignment at a minimum total cost.

[1]The term "critical edge" stems from compiler algorithms (e.g., partial redundancy elimination) that prohibit this kind of edge. Critical edges are split, i.e., nodes are inserted on critical edges, to convert them to non-critical edges.

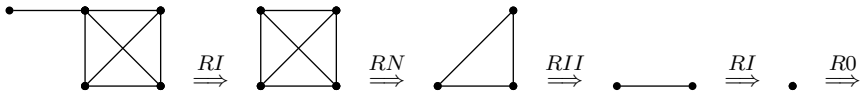


Fig. 1: Reduction sequence.

We solve the PBQP problem using matrix notation. A discrete variable x_i becomes a Boolean vector $\vec{x}_i$ whose vector elements are zeros and ones, and whose length is determined by the number of elements in its domain $\mathbb{D}_i$. Each 0-1 element of $\vec{x}_i$ corresponds to an element of $\mathbb{D}_i$. An assignment of x_i to d_i is represented as a unit vector whose element for d_i is set to one. Hence, a possible assignment for a variable x_i is modeled by the constraint $\vec{x}_i\vec{1}^T = 1$ that restricts vectors $\vec{x}_i$ such that only one vector element is assigned one and all other elements are set to zero.

The cost function $C(x_i, x_j, d_i, d_j)$ is decomposed for each pair (x_i, x_j). The costs for the pair are represented by matrix $\mathcal{C}_{ij}$. A matrix element corresponds to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is mapped to cost vectors[2] $\vec{c}_i$. Quadratic forms and scalar products are employed to formulate PBQP as a mathematical program:

$$\begin{aligned}\text{minimize } f = &\left[\sum_{1\le i\le n} \vec{c}_i\vec{x}_i^T\right] + \left[\sum_{1\le i<j\le n} \vec{x}_i\mathcal{C}_{ij}\vec{x}_j^T\right]\\ \text{subject to: } &\forall 1 \le i \le n : \vec{x}_i \in \{0,1\}^{|\mathbb{D}_i|}\\ &\forall 1 \le i \le n : \vec{x}_i\vec{1}^T = 1.\end{aligned}$$

In [Scholz and Eckstein 2002; Eckstein 2003] a solver was introduced, which solves a sub-class of PBQP problems optimally in $\mathcal{O}(nm^3)$, where n is the number of discrete variables and m is the maximal number of elements in their domains, i.e., $m = \max(|\mathbb{D}_1|, \ldots, |\mathbb{D}_n|)$. The solver uses an undirected graph called PBQP graph as the underlying data structure. The nodes of the PBQP graph are the discrete variables x_i, for all i $(1 \le i \le n)$. In the graph there exists an edge (x_i, x_j) if the cost function $C(x_i, x_j, h(x_i), h(x_j))$ is not equal to zero for an arbitrary solution of h, i.e., matrix $\mathcal{C}_{ij}$ is not zero. An instance of the PBQP problem is regarded as trivial if the nodes in the PBQP graph are disconnected, i.e., there are no edges in the PBQP graph.

The solver uses reductions to solve an instance of a PBQP problem. A reduction maps an instance of the PBQP problem to a new instance that has one less discrete variable. If the solution for this new problem instance is known, the solution for the eliminated discrete variables can be computed. Hence, the solver has two phases: (1) a reduction phase that eliminates variables until a trivial instance (the empty graph) of the PBQP problem remains, and (2) a back propagation phase that determines the solutions for the eliminated variables.

The solver employs the reductions R0, RI, and RII to eliminate discrete variables whose nodes in the PBQP graph have either a node degree of zero, one or two. Figure 1 depicts the reduction of a PBQP graph. If none of the above reductions can be applied (as shown in Figure 1), the problem becomes irreducible and a

[2]Note that vectors are row vectors in our notation.

heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable $\vec{x}_i$ and an assignment by searching for local minima. The solution obtained is guaranteed to be optimal if the reduction RN is not applied [Eckstein 2003]. Even if there are RN nodes in the PBQP graph, an optimal solution can be obtained by branch-and-bound techniques [Hames and Scholz 2006].

3. MOTIVATION

The goal of our optimization is to insert the *minimal* number of bank selection instructions while ensuring that the banked memory is accessed correctly. The underlying optimization assumptions are that all variables in a program have been assigned to memory banks and that our optimization does not re-order statements to further minimize the number of bank selection instructions. For the sake of simplicity, we assume that a statement has at most one banked-memory access. To extend the optimization to more than one banked-memory access per statement, the optimization is performed for each bank register separately.

A statement is said to be *bank-sensitive* if it accesses banked memory, otherwise it is *transparent*. For example, all banked-memory accesses of load and store statements are bank-sensitive. A bank-sensitive statement requires that the appropriate bank is made active prior to its execution. Otherwise, the program semantics are violated.

In the intra-procedural optimization, function calls are considered to be bank-sensitive but are handled differently from load and store statements. For a function call, we do not know which bank is active upon return. Therefore, a call statement denotes a banked-memory access to an unknown bank. To optimize bank selection instructions across call sites, an extension of the intra-procedural optimization is described in Section 4.4.

We define a local predicate $bank(s)$ that indicates the bank property of statement s:

$$bank(s) = \begin{cases} b_*, & \text{if } s \text{ is transparent,} \\ b_x, & \text{if } s \text{ requires bank } b_x, \\ b_?, & \text{if } s \text{ requires an unknown bank.} \end{cases} \tag{2}$$

For a bank-sensitive statement, $bank(s)$ is either $b_?$ denoting an unknown bank, or b_x denoting a concrete bank.

A linear scan over a basic block is sufficient to find a minimal placement of bank selection instructions in the basic block. However, with a linear scan it is not possible to determine whether the bank of the first bank-sensitive statement is already active at the entry of the basic block. Therefore, placing bank selection instructions for the first bank-sensitive statements becomes an intra-procedural optimization problem.

If a basic block has only transparent statements then we call it a *transparent basic block*, otherwise it is *bank-sensitive*. In our intraprocedural analysis, we need to distinguish between *transparent* basic blocks $u \in T$ and *bank-sensitive* basic blocks $u \in S$, where T is the set of transparent basic blocks and S is the set of bank-sensitive basic blocks.

The bank selection instruction for the first bank-sensitive statement is the only bank selection instruction that can be beneficially moved across basic basic block

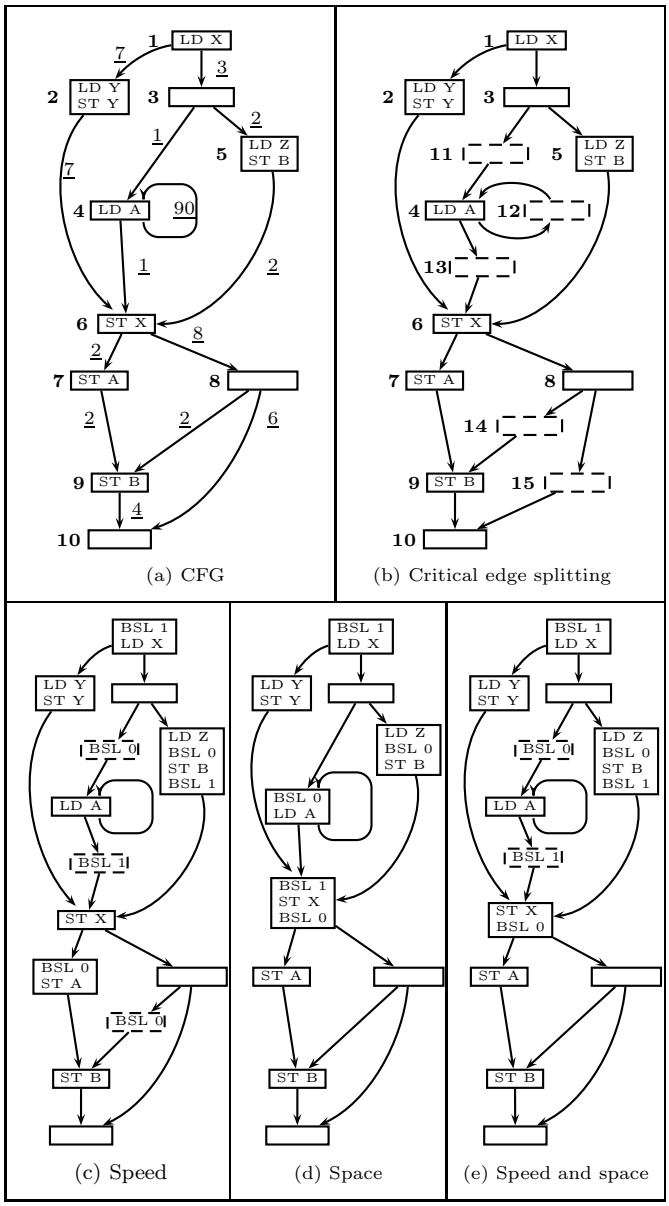


Fig. 2: An example for a two-bank architecture (`A` and `B` reside in bank 0, `X`, `Y` and `Z` in bank 1).

boundaries. Hence, the transformation of the intraprocedural optimization limits the placement of bank selection instructions to the following points inside a basic block: (1) before the first bank-sensitive statement, (2) after the last bank-sensitive statement, and (3) inside a transparent basic block.

Splitting a critical edge creates a transparent basic block (aka. *critical basic block*) in the CFG. Critical basic blocks are potential hosts for bank selection instructions and therefore yield optimization opportunities. However, splitting a critical edge is not free because an additional jump statement must be inserted. Overhead costs and optimization opportunities of critical basic blocks are considered during cost analysis of our optimization. A critical edge is only split if the cost analysis of the intra-procedural optimization decides to place a bank selection instruction in the resulting critical basic block. Otherwise the critical edge is kept (with zero overhead cost).

Consider our running example in Figure 2(a). Basic blocks are numbered in **bold**, execution frequencies are denoted by underlined numbers on edges. The execution frequency of a basic block (except the start node) is the sum of frequencies of its incoming edges. For the start node it is the sum of frequencies of its outgoing edges. Figure 2(b) shows the CFG where all five critical edges have been split tentatively. Let us assume that our example architecture has two banks, i.e., bank 0 and bank 1, and either bank 0 or bank 1 is active. All memory operations are done by load and store statements of the form `LD` v and `ST` v, where v is a variable residing in either bank 0 or bank 1. Our example has five variables: `A` and `B` reside in bank 0, and `X`, `Y` and `Z` in bank 1. Before a load or store for variable v is executed, the bank of the variable must be active.

A naive approach to ensure correct code is to issue a bank selection instruction prior to all banked-memory accesses. However, this approach produces sub-optimal code. For example, basic block 4 that contains `LD A` inside a loop would require the bank selection instruction `BSL 0`.

Figures 2(c)–(e) illustrate the minimal solutions that we find with respect to the three optimization criteria: speed, space, and a combination of speed and space. In our cost model, we take into account the costs of additional jump statements introduced in critical basic blocks (critical basic blocks are shown as dashed boxes in Figure 2(b)). We assume that bank selection instructions and jump statements have an instruction length of one byte and they take one cycle to execute. If we want to minimize the number of bank selection instructions inserted, we can measure the cost of inserting a bank selection instruction in a basic block as the dynamic number of cycles spent on executing the bank selection instruction times the execution frequency of the basic block. If we place the bank selection instruction in a critical basic block, we need to add the extra cost for the jump statement. The minimal solution for speed is shown in Figure 2(c) where we place a bank selection instruction before and after the loop and in basic block 14. In Figure 2(d) the optimization for space is shown. `BSL 0` stays inside the loop to avoid the additional jump statement required if `BSL 0` is placed in critical basic block 11. Optimizing for space reduces the memory footprint but increases the execution time of the program. An optimization which combines speed and space objectives is shown in Figure 2(e). For this particular example the speed objective was weighted one third

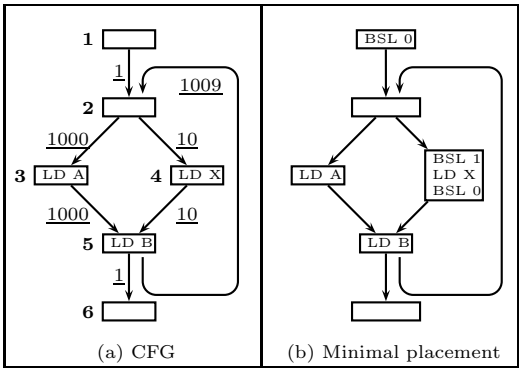


Fig. 3: Minimal placement by bank selection optimization but not by SPRE (variables `A` and `B` are assigned to bank 0 and variable `X` to bank 1).

and the space objective was weighted two thirds.

3.1 Bank Selection Optimization vs. SPRE

The goal of Speculative Partial Redundancy Elimination (SPRE) is to minimize the dynamic number of computations for a given expression in a CFG with respect to an edge profile [Cai and Xue 2003; Scholz et al. 2004]. Computations of expressions are moved from heavily executed paths to rarely executed paths and computations are performed speculatively, i.e., without a guarantee that computations are used later. For SPRE, expressions are considered to be independent from each other, and the expression order of the SPRE transformation is arbitrary.

A bank selection instruction for a specific bank could be seen as a computation in SPRE. However, the placement of bank selection instructions for a specific bank is not independent because bank selection instructions modify the same bank register. Hence, a necessary assumption of SPRE is violated and SPRE cannot solve the minimal placement of bank selection instructions. While SPRE may be used to find the minimal placement for the set of all instructions that access a specific bank, the solutions obtained can lead to incorrect or sub-optimal code.

An example is illustrated in Figure 3. For the CFG given in Figure 3(a), the minimal placement found by our bank selection formulation is shown in Figure 3(b). If SPRE is used instead, one may proceed as follows. For the first bank-sensitive instruction in every basic block, the required bank selection instruction is inserted just before the instruction. As a result, some of these instructions may be partially or fully redundant. In the current example, `BSL 0` will be inserted at the entries of basic blocks 3 and 5, and `BSL 1` at the entry of basic block 4. The three instructions are all partially redundant (in the dynamic sense). To eliminate such redundancies, all distinct bank selection instructions are treated as distinct expressions. If there are m banks, there will be m SPRE problems, one for each distinct bank. There are two approaches to solving these m SPRE problems. First, they are solved independently of each other. But this can lead to incorrect code. In the current example, `BSL 0` and `BSL 1` will both be inserted inside basic block 1 so that one of the two instructions ends up being removed effectively but incorrectly. Second,

```
OptimizeBasicBlock(u)
1   first ← entry(u)
2   last ← entry(u)
3   rbank ← b*
4   for all s ∈ u in seq. order do
5     if bank(s) ≠ b* then
6       if rbank = b* then
7         first ← s
8       else
9         if bank(s) ≠ b? ∧ bank(s) ≠ rbank then
10          insert BSL ⟨bank(s)⟩ before s
11        endif
12      endif
13      rbank ← bank(s)
14      last ← s
15    endif
16  endfor
17  return(first, last)
```

Fig. 4: Local optimization.

the m SPRE problems are solved one after another by considering the effects of the bank selection instructions introduced when the earlier SPRE problems are solved. However, this second approach also fails since there does not exist a linear order to guarantee the minimal placement. In the current example, the minimal solution can be found if we process `BSL 0` and then `BSL 1`. It is easy to see that the order should be swapped if edge (2,4) is far more frequently executed than edge (2,3). Since both scenarios can co-exist in the same CFG, SPRE cannot determine the minimal placement of bank selection instructions. Instead, the minimal placement must be sought when all bank selections are considered together. This is the major contribution of this paper.

4. BANK SELECTION OPTIMIZATION

We develop the bank selection optimization in four steps. In the first step we discuss how to optimize bank selection instructions inside a basic block. In the second step we formulate the intraprocedural optimization as a discrete optimization problem. In the third step we show how the discrete optimization problem is mapped to the PBQP problem, and the last step extends the intraprocedural optimization to whole programs.

4.1 Local Optimization

Given a basic block in which all variables have been assigned to banks, this section gives an algorithm that minimizes the number of bank selection instructions inserted in the basic block.

In Figure 4, the linear scan algorithm for inserting bank selection instructions is listed. The algorithm initializes variable *first*, which points to the first bank-sensitive statement; variable *last*, which points to the last bank-sensitive statement;

Basic Block		*bank*(s)	*first*	*last*
s_0	`NOP`	b_*	s_0	s_0
s_1	`LD X`	1	s_1	s_1
	`BSL 0`	inserted		
s_2	`ST A`	0	s_1	s_2
s_3	`CALL foo`	$b_?$	s_1	s_3
	`BSL 0`	inserted		
s_4	`LD B`	0	s_1	s_4
	`BSL 1`	inserted		
s_5	`ST Y`	1	s_1	s_5
s_6	`ST Z`	1	s_1	s_6
s_7	`NOP`	b_*	s_1	s_6

Fig. 5: Example: variables A and B reside in bank 0, and variables X, Y, and Z in bank 1.

and variable *rbank*, which represents the active bank of the bank register. Inside the loop, we ignore transparent statements, i.e., those statements s that satisfy $bank(s) = b_*$ (in line 5). When the first bank-sensitive statement is reached, the algorithm sets variable *first*. For all subsequent bank-sensitive statements, the algorithm checks whether a new bank selection instruction needs to be issued. This is the case if the required bank $bank(s)$ for statement s is not $b_?$ and differs from the bank required by a preceding bank-sensitive statement. After having traversed the basic block, the algorithm returns the first and last bank-sensitive statements of the basic block. If the basic block is transparent, *first* and *last* will point to the first statement of the basic block (due to lines 1 and 2), and the following holds: $bank(first) = b_*$ and $bank(last) = b_*$.

Given a basic basic block u, we write $first(u)$ and $last(u)$ to denote the *first* and *last* statement returned by the algorithm in Figure 4. We use fb_u to denote the bank of statement $first(u)$, and lb_u to denote the bank of statement $last(u)$.

Figure 5 illustrates the operation of our linear scan algorithm on a sample basic block consisting of the instructions s_0–s_7. In this example we assume that variables `A` and `B` reside in bank 0, and variables `X`, `Y` and `Z` reside in bank 1. The column entitled "$bank(s)$" depicts the bank-sensitivity of the respective statements. We use NOP statements to introduce transparency to this example. The call to function `foo` in statement s_3 potentially modifies the bank register. Therefore, the bank of this statement is unknown ($b_?$). Columns "*first*" and "*last*" denote the first and last bank-sensitive statement as the algorithm progresses through the basic block. Since statement s_0 is transparent, *first* is eventually set to statement s_1. For each bank-sensitive statement, *last* is updated. The linear scan algorithm introduces three `BSL` instructions and sets *first* and *last* to the first and last bank-sensitive statement of the basic block.

Note that the optimization algorithm for basic blocks does not insert a bank selection instruction for the first bank-sensitive statement. If we do not take into account the intra-procedural flow across basic blocks in our analysis, we could insert the bank selection instruction prior to the first bank-sensitive statement. However, this would result in a sub-optimal solution for the entire program.

Table I. Transformation for configuration (P_u, Q_u).

For a bank-sensitive basic block:

Location	Insertion	Condition
entry	$\texttt{BSL}\langle fb_u \rangle$	$fb_u \neq b_? \wedge P_u \neq fb_u$
exit	$\texttt{BSL}\langle Q_u \rangle$	$Q_u \neq b_? \wedge Q_u \neq lb_u$

For a transparent basic block:

Insertion	Condition
$\texttt{BSL}\langle Q_u \rangle$	$Q_u \neq b_? \wedge P_u \neq Q_u$

4.2 Intra-Procedural Optimization

The intra-procedural optimization is effective because bank selection instructions can be hoisted across basic blocks. For example, instead of performing the bank selection instruction for `LD A` of Figure 2 inside the loop, we move the bank selection instruction outside of the loop when optimizing for speed (as depicted in Figure 2(c)).

Our approach uses discrete optimization to place bank selection instructions. The main idea is that we introduce two *controlling variables* P_u and Q_u for every basic block u. These two variables describe the state of the bank register before and after execution of the basic block. The domain of P_u and Q_u is $\mathbb{D} = \{b_0, \ldots, b_{m-1}, b_?\}$, i.e., variables P_u and Q_u are either set to a concrete bank, or the state of the bank register is unknown. The semantics of the controlling variables are as follows: If P_u is set to b_x, we can assume that the bank register has been set to b_x prior to the execution of basic block u. If P_u is set to $b_?$, then the state of the bank register is unknown upon entry of u. Conversely, variable Q_u forces basic block u to guarantee that the bank register is set to Q_u upon exit.

Depending on the values of P_u and Q_u we insert bank selection instructions according to Table I. For a bank-sensitive basic block, there are at most two insertions, i.e., one before the first bank-sensitive statement and one after the last bank-sensitive statement. The first insertion ensures that the bank register is set to bank fb_u if variable P_u is not set to this bank. The second insertions is used to guarantee that the bank register is set to Q_u after executing basic block u. For a transparent basic block at most one bank selection instruction is inserted to ensure that the bank register is set to Q_u upon exit (cf. again Table I).

A *bank selection transformation* $\mathbb{T} \in (\mathbb{D} \times \mathbb{D})^{|V|}$ is defined by configurations (P_u, Q_u) for all basic blocks u. All possible insertions of bank selection instructions at entries and exits of basic blocks are covered by at least one configuration of a basic block.

A bank selection transformation $\mathbb{T}$ is *correct* if controlling variable P_s of the entry node is unknown and for all CFG edges (u, v) it holds that

$$(P_v \neq b_?) \Rightarrow (Q_u = P_v)\,. \tag{3}$$

The start node s cannot assume that a specific bank is active prior to its execution. Therefore, we set P_s to $b_?$. For all other nodes, each predecessor needs to have bank Q_u active upon exit if P_v is not equal to the unknown bank.

The controlling variables P_u and Q_u determine the correctness of a transformation and its costs. For each basic block u, we have a cost function $cost_u(P_u, Q_u)$ that returns the costs for a given configuration (P_u, Q_u). These costs are chosen from arbitrary metrics such as speed, space, and mixed cost models.

A bank selection transformation $\mathbb{T}$ is minimal if it is correct and if the costs of a

transformation are minimal:

$$\min f = \sum_{u \in V} cost_u(P_u, Q_u). \tag{4}$$

We split the cost function into the costs for bank-sensitive basic blocks ($u \in S$) and transparent basic blocks ($u \in T$).

$$f = \sum_{u \in V} cost_u(P_u, Q_u) = \sum_{u \in S} s\text{-}cost_u(P_u, Q_u) + \sum_{u \in T} t\text{-}cost_u(P_u, Q_u) \tag{5}$$

Without loss of generality we divide the costs for a bank-sensitive basic block into the costs occurring upon entry and exit of the basic block. Function $n\text{-}cost_u(P_u)$ accounts for the cost of the bank selection instruction at the entry of the basic block and function $e\text{-}cost_u(Q_u)$ upon exit:

$$s\text{-}cost_u(P_u, Q_u) = n\text{-}cost_u(P_u) + e\text{-}cost_u(Q_u). \tag{6}$$

Both functions are zero if no insertion is performed. Otherwise they return the cost c_u of a bank selection instruction.

$$n\text{-}cost_u(P_u) = \begin{cases} c_u, & \text{if } fb_u \neq b_? \wedge P_u \neq fb_u \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$e\text{-}cost_u(Q_u) = \begin{cases} c_u, & \text{if } Q_u \neq b_? \wedge Q_u \neq lb_u \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

For a transparent basic block the costs are defined by

$$t\text{-}cost_u(P_u, Q_u) = \begin{cases} c_u, & \text{if } Q_u \neq b_? \wedge P_u \neq Q_u \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

In Eqns. (7) – (9), constant c_u represents the insertion cost of a bank selection instruction in basic block u. Costs are computed based on a chosen metric. The only restriction we impose on such a metric is that the costs must have positive values[3], i.e., $c_u \geq 0$. In our experiment, we have chosen the parameterizable cost metric

$$c_u = \alpha \times \mathsf{SPEED_u} + \beta \times \mathsf{SPACE_u}, \tag{10}$$

with its two parameters α and β controlling the weights of the speed and space objectives. Depending on whether u is a critical basic block, $\mathsf{SPEED_u}$ is defined as

$$\mathsf{SPEED_u} = \begin{cases} (\text{bsl-cycles} + \text{jump-cycles}) \times \text{freq}_u, & \text{if } u \text{ is critical} \\ \text{bsl-cycles} \times \text{freq}_u, & \text{otherwise.} \end{cases} \tag{11}$$

Therein bsl-cycles denotes the number of cycles taken for executing one single bank selection instruction, jump-cycles is the number of cycles taken for executing one additional (unconditional) jump statement introduced in basic block u due to edge splitting, and freq_u is the execution frequency of u (obtained by profiling).

[3]Note that the PBQP solver would also cope with negative costs. However, in the context of bank selection negative costs are not sensible.

Similarly, the constant $\mathsf{SPACE_u}$ is defined as

$$\mathsf{SPACE_u} = \begin{cases} \text{bsl-size} + \text{jump-size}, & \text{if } u \text{ is critical} \\ \text{bsl-size}, & \text{otherwise}, \end{cases} \tag{12}$$

where bsl-size (jump-size) is the size of a bank selection (jump) instruction (measured in bytes).

Putting all constraints and costs together, we formulate the intra-procedural bank selection problem as the following discrete optimization:

$$\begin{aligned} &\text{s.t.} \, \forall u \in V : P_u, Q_u \in \mathbb{D} \\ &\quad P_s = b_? \\ &\quad \forall (u,v) \in E : (P_v \neq b_?) \Rightarrow (Q_u = P_v) \\ &\min f = \sum_{u \in S} n\text{-}cost_u(P_u) + \sum_{u \in S} e\text{-}cost_u(Q_u) + \sum_{u \in T} t\text{-}cost_u(P_u, Q_u). \end{aligned} \tag{13}$$

A related problem was introduced in [Kleinberg and Tardos 1999], which is a classification problem and it was shown that this problem is hard to solve, i.e., a set of points should be labeled such that a cost function is to be minimized. The cost function takes into account costs for local labeling and labeling of two related points. In [Kleinberg and Tardos 1999] an approximation algorithm was introduced. However, the approximation algorithm is not practical. Instead, we use the PBQP problem to solve the underlying discrete optimization problem for bank selection, for which we have a very efficient and effective solver.

4.3 Mapping to PBQP

We employ PBQP [Scholz and Eckstein 2002; Eckstein 2003] to solve the discrete optimization problem of Eqn. (13). The controlling variables P_u and Q_u become Boolean vector variables $\vec{p}_u$ and $\vec{q}_u$. The elements of the vectors represent an element in $\mathbb{D} = \{b_0, \ldots, b_{m-1}, b_?\}$. PBQP restricts $\vec{p}_u$ and $\vec{q}_u$ to Boolean vectors whose elements are set to zero except one element is set to one, i.e. variables P_u and Q_u have a concrete bank assignment or they are set to $b_?$.

The costs of bank-sensitive basic blocks are modeled as scalar products and the costs of transparent basic blocks become quadratic forms. The objective function of Eqn. (13) is mapped to the PBQP objective function

$$f = \sum_{u \in S} \vec{n}_u \vec{p}_u^T + \sum_{u \in S} \vec{e}_u \vec{q}_u^T + \sum_{u \in T} \vec{p}_u (c_u \cdot \mathcal{T}) \vec{q}_u^T. \tag{14}$$

Therein $\vec{n}_u \vec{p}_u^T$ and $\vec{e}_u \vec{q}_u^T$ are the cost functions $n\text{-}cost_u(P_u)$ and $e\text{-}cost_u(Q_u)$ in vector notation, and $\vec{p}_u (c_u \cdot \mathcal{T}) \vec{q}_u^T$ is $t\text{-}cost_u(P_u, Q_u)$ as a quadratic form. The vector $\vec{n}_u$ in scalar product $\vec{n}_u \vec{p}_u^T$ is a zero vector if the bank of the first bank-sensitive statement is $b_?$, otherwise it is the vector

$$\begin{array}{c|cccccccccc} & b_0 & b_1 & \ldots & b_{i-1} & b_i & b_{i+1} & \ldots & b_{m-1} & b_? \\ \hline \vec{n}_u & c_u & c_u & \ldots & c_u & 0 & c_u & \ldots & c_u & c_u \end{array}, \tag{15}$$

where b_i is bank fb_u of the first bank-sensitive statement and c_u are the costs for inserting a bank selection instruction (i.e., `BSL` b_i). Cost function $e\text{-}cost(Q_u)$

Table II. Cost matrices.

$\mathcal{T}$	b_0	b_1	...	...	b_{m-1}	$b_?$
b_0	0	1	...	...	1	0
b_1	1	0	1	...	1	0
⋮	⋮	⋱	⋱	⋱	⋮	⋮
b_{m-2}	1	...	1	0	1	0
b_{m-1}	1	...	...	1	0	0
$b_?$	1	...	...	...	1	0

(a) Transparent basic blocks

$\mathcal{R}$	b_0	b_1	...	...	b_{m-1}	$b_?$
b_0	0	∞	...	...	∞	0
b_1	∞	0	∞	...	∞	0
⋮	⋮	⋱	⋱	⋱	⋮	⋮
b_{m-2}	∞	...	∞	0	∞	0
b_{m-1}	∞	...	...	∞	0	0
$b_?$	∞	...	...	∞	∞	0

(b) Correctness constraint for CFG edges

becomes the scalar product $\vec{e}_u\vec{q}_u^T$, for which the cost vector $\vec{e}_u$ is

$$\begin{array}{c|cccccccc} & b_0 & b_1 & \dots & b_{j-1} & b_j & b_{j+1} & \dots & b_{m-1} & b_? \\ \hline \vec{e}_u & c_u & c_u & \dots & c_u & 0 & c_u & \dots & c_u & 0 \end{array}. \tag{16}$$

Therein bank b_j denotes bank lb_u, and we do not issue a bank selection instruction if Q_u is equal to b_j or $b_?$.

For example, consider basic block 4 in our motivating example (see Figure 2(a)). This basic block is executed 91 times. The first bank-sensitive statement is `LD A`, which accesses bank b_0. If we assume that it takes one cycle to execute a bank selection instruction, then we have 91 cost units for the insertion before `LD A`. Thus $c_u = 91$ and cost function *n-cost*(P_4) becomes $\vec{n}_4 = \begin{pmatrix} 0 & 91 & 91 \end{pmatrix}$. The first element of the vector imposes zero costs if P_4 is set to b_0. If P_4 is set to b_1 or $b_?$, 91 cost units are imposed because a bank selection instruction needs to be inserted prior to `LD A`. Cost vector $\vec{e}_4$ is $\begin{pmatrix} 0 & 91 & 0 \end{pmatrix}$, because there are zero costs imposed for Q_4 equal to b_0 or $b_?$.

The transparent cost function *t-cost*$_u(P_u, Q_u)$ is expressed as quadratic form $\vec{p}_u(c_u\mathcal{T})\vec{q}_u^T$. Matrix $\mathcal{T}$ is given in Table II(a) and it is generated based on Eqn. (9). The rows of $\mathcal{T}$ correspond to variable P_u and the columns correspond to variable Q_u. If P_u and Q_u are not set to the same bank and Q_u is not $b_?$, one cost unit is imposed (zero costs otherwise). We multiply matrix $\mathcal{T}$ with scalar c_u in Eqn. (14) to model the actual insertion costs for a bank selection instruction.

To enforce correct transformations, we use the standard technique of encoding the correctness constraint as part of the objective function, which we extend to $g = f + \Delta$, where Δ is 0 if the transformation $\mathbb{T}$ is a correct transformation, and ∞ otherwise. The correctness constraints defined over CFG edges are mapped to a sum of a scalar product and quadratic forms,

$$\Delta = \begin{pmatrix} \infty & \dots & \infty & 0 \end{pmatrix} \vec{q}_s^T + \sum_{(u,v)\in E} \vec{q}_u \mathcal{R} \vec{p}_v^T, \tag{17}$$

where the constraint expressed in Eqn. (3) is mapped to matrix $\mathcal{R}$ shown in Table II(b) and the constraint to set P_s to $b_?$ is mapped to a scalar product. Quadratic forms are used to express the correctness constraints. In matrix $\mathcal{R}$ the diagonal and the last column contain zeroes, representing the cases where Q_u is equal to P_v or where P_v is set to $b_?$. All other assignments of Q_u and P_v are penalized with ∞ costs.

Our running example in Figure 2 assumes an architecture with two banks. Hence,

we get the following correctness matrix $\mathcal{R}$ and transparent matrix $\mathcal{T}$.

$\mathcal{R}$	b_0	b_1	$b_?$
b_0	0	∞	0
b_1	∞	0	0
$b_?$	∞	∞	0

$\mathcal{T}$	b_0	b_1	$b_?$
b_0	0	1	0
b_1	1	0	0
$b_?$	1	1	0

The objective function without correctness constraints is

$$\begin{aligned} f = &\vec{n}_1\vec{p}_1^{\mathcal{T}} + \vec{e}_1\vec{q}_1^{\mathcal{T}} + \vec{n}_2\vec{p}_2^{\mathcal{T}} + \vec{e}_2\vec{q}_2^{\mathcal{T}} + \vec{n}_4\vec{p}_4^{\mathcal{T}} + \vec{e}_4\vec{q}_4^{\mathcal{T}} + \vec{n}_5\vec{p}_5^{\mathcal{T}} + \vec{e}_5\vec{q}_5^{\mathcal{T}} + \vec{n}_6\vec{p}_6^{\mathcal{T}} + \vec{e}_6\vec{q}_6^{\mathcal{T}} + \\ &\vec{n}_7\vec{p}_7^{\mathcal{T}} + \vec{e}_7\vec{q}_7^{\mathcal{T}} + \vec{n}_9\vec{p}_9^{\mathcal{T}} + \vec{e}_9\vec{q}_9^{\mathcal{T}} + \vec{p}_3(c_3 \cdot \mathcal{T})\vec{q}_3^{\mathcal{T}} + \vec{p}_8(c_3 \cdot \mathcal{T})\vec{q}_8^{\mathcal{T}} + \vec{p}_{10}(c_{10} \cdot \mathcal{T})\vec{q}_{10}^{\mathcal{T}} + \\ &\vec{p}_{11}(c_{11} \cdot \mathcal{T})\vec{q}_{11}^{\mathcal{T}} + \vec{p}_{12}(c_{12} \cdot \mathcal{T})\vec{q}_{12}^{\mathcal{T}} + \vec{p}_{13}(c_{13} \cdot \mathcal{T})\vec{q}_{13}^{\mathcal{T}} + \vec{p}_{14}(c_{14} \cdot \mathcal{T})\vec{q}_{14}^{\mathcal{T}} + \\ &\vec{p}_{15}(c_{15} \cdot \mathcal{T})\vec{q}_{15}^{\mathcal{T}}, \end{aligned}$$

where c_u for basic blocks 3, 8, 10, 11, 12, 13, 14 and 15 denotes the costs for inserting bank selection instructions. Constants c_u are dependent on the optimization criteria.

The correctness constraints are

$$\begin{aligned} \Delta = &\begin{pmatrix}\infty & \infty & 0\end{pmatrix}\vec{p}_s^{\mathcal{T}} + \vec{q}_1\mathcal{R}\vec{p}_2^{\mathcal{T}} + \vec{q}_1\mathcal{R}\vec{p}_3^{\mathcal{T}} + \vec{q}_2\mathcal{R}\vec{p}_6^{\mathcal{T}} + \vec{q}_3\mathcal{R}\vec{p}_5^{\mathcal{T}} + \vec{q}_3\mathcal{R}\vec{p}_{11}^{\mathcal{T}} + \vec{q}_4\mathcal{R}\vec{p}_{12}^{\mathcal{T}} + \\ &\vec{q}_4\mathcal{R}\vec{p}_{13}^{\mathcal{T}} + \vec{q}_5\mathcal{R}\vec{p}_6^{\mathcal{T}} + \vec{q}_6\mathcal{R}\vec{p}_7^{\mathcal{T}} + \vec{q}_6\mathcal{R}\vec{p}_8^{\mathcal{T}} + \vec{q}_7\mathcal{R}\vec{p}_9^{\mathcal{T}} + \vec{q}_8\mathcal{R}\vec{p}_{14}^{\mathcal{T}} + \vec{q}_8\mathcal{R}\vec{p}_{15}^{\mathcal{T}} + \vec{q}_9\mathcal{R}\vec{p}_{10}^{\mathcal{T}} + \\ &\vec{q}_{11}\mathcal{R}\vec{p}_4^{\mathcal{T}} + \vec{q}_{12}\mathcal{R}\vec{p}_4^{\mathcal{T}} + \vec{q}_{13}\mathcal{R}\vec{p}_6^{\mathcal{T}} + \vec{q}_{14}\mathcal{R}\vec{p}_9^{\mathcal{T}} + \vec{q}_{15}\mathcal{R}\vec{p}_{10}^{\mathcal{T}}. \end{aligned}$$

The objective function $g = f + \Delta$ is to be solved to find the minimal bank selection placement for the running example.

4.4 Inter-procedural Optimization

The bank selection optimization can be extended to hoist bank selection instructions across call sites. For frequently executed calls the inter-procedural optimization is highly effective. The inter-procedural transformation extends the placement of bank selection instructions to the following program points: (1) at the entry of a subroutine, (2) at the exit of a subroutine, (3) and before a call. In contrast to the intra-procedural optimization, a single discrete optimization problem solves the bank selection of the whole program in one step.

The mathematical model of the inter-procedural optimization problem is an extension of the intra-procedural approach. The discrete optimization problem for each subroutine is constructed as outlined in the previous section. However, the boundary condition for the start node is removed except for the main subroutine of the program. Each call site l_i of subroutine F becomes a basic block of its own (with controlling variables P_{l_i} and Q_{l_i}). Correctness constraints between the call site and the callee ensure a consistent state of the bank register upon procedure call and return. The correctness constraints between the callers and the callee are depicted in Figure 6 (edges represent correctness constraints). For call site l_i we impose a correctness constraint between discrete variable P_{l_i} and P_s such that

$$(P_s \neq b_?) \Rightarrow (P_s = P_{l_i}) \tag{18}$$

holds. For the end node of a subroutine we add correctness constraint

$$(Q_{l_i} \neq b_?) \Rightarrow (Q_{l_i} = Q_e)\,. \tag{19}$$

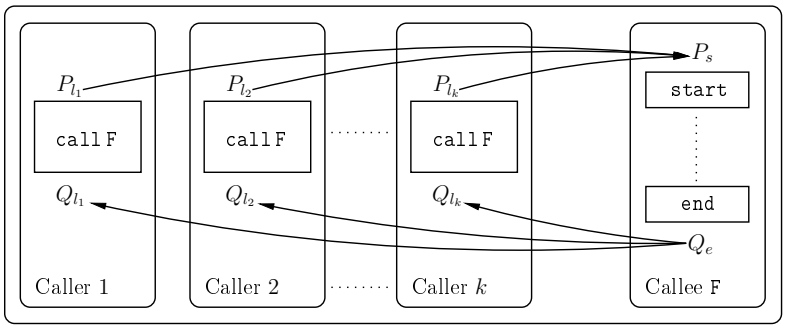


Fig. 6: Interprocedural analysis.

The correctness constraints ensure that the call site sets the correct bank if the bank selection is not performed for the first bank-sensitive statement within subroutine F. If the statements after the call expect a certain bank to be set by the subroutine, a bank selection instruction is inserted after the last bank-sensitive statement of subroutine F.

The PBQP problem is extended for all subroutines F in the program and for all calls $\{x_1, \ldots, x_k\}$ of the subroutine F as given below:

$$\Delta' = \Delta + \vec{p}_{x_i} \mathcal{R} \vec{p}_s^T + \vec{q}_e \mathcal{R} \vec{q}_{x_i}^T. \tag{20}$$

The correctness constraint is identical to Eqn. (3) and therefore the same matrix $\mathcal{R}$ as in the intra-procedural mapping is used to map the discrete optimization problem to PBQP.

4.5 Non-Uniform Cost Models

Architectures such as the PIC 16Fxxx microcontroller do not have uniform costs for bank switching. Instead of having a single bank selection instruction with constant execution speed and program space costs, the PIC 16Fxxx architecture provides only bit access to the bank register; depending on the previously active bank the bank switching costs may vary. For example, assume we want to set the bank register to bank 1. For this task we need to issue two assembly instructions.

```
bsf STATUS, RP0
bcf STATUS, RP1
```
(21)

These assembly instructions set the first and second bit of the bank register to one and zero, where `RP0` and `RP1` denote the first and second bit of the bank register [Microchip Technology Inc. 2003]. If we know that the first bit of the bank register is already set to one, the first instruction can be omitted and the number of bank selection instructions for setting bank 1 is halved.

The optimization benefits from the knowledge of the state of the bank register before switching. Since the state of the bank register is reflected in the discrete variables P_u and Q_u at the entry and exit of a basic block, it would be straightforward to extend the cost functions for bank-sensitive basic blocks and transparent basic blocks. However, there are cases where the state of the bank register can only be partially deduced. As an example, assume the CFG fragment depicted in

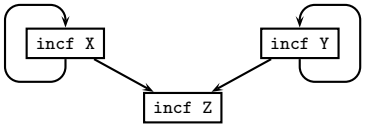


Fig. 7: Example: Variables `X`, `Y` and `Z` are in banks 1,3 and 1. Instruction `incf` increments the value of its operand.

Figure 7, with variables `X`, `Y` and `Z` residing in banks 1, 3 and 1. Let us assume that the nodes preceding the confluence node have higher execution frequencies. Hence, it is beneficial to place the bank selection instruction in the basic block of `incf Z`. The bank register is either set to bank 1 or bank 3 prior to execution of `incf Z`. At `incf Z` the second bit `RP1` is unknown, but the first bit `RP0` is known to be one. Hence, the instruction `bcf STATUS, RP1` must be issued prior to the execution of `incf Z`, but the instruction `bsf STATUS, RP0` can be omitted. It should be noted that data flow analysis is able to deduce the bits of the bank register, however, it cannot solve the underlying discrete optimization problem.

To model the unknown bits in the discrete domain $\mathbb{D}$ of variables P_u and Q_u, we use a bit representation of the banks

$$\mathbb{D} = \left\{ b_x : x \in \{0, 1, ?\}^k \right\}, \tag{22}$$

where k is the bit length of the bank register, i.e., $m = 2^k$. For example, the bank register of the PIC 16Fxxx architecture has a bit length of 2. Therefore, the discrete domain is given by

$$\mathbb{D} = \{b_{\langle 0,0\rangle}, b_{\langle 0,1\rangle}, b_{\langle 1,0\rangle}, b_{\langle 1,1\rangle}, b_{\langle ?,0\rangle}, b_{\langle ?,1\rangle}, b_{\langle 0,?\rangle}, b_{\langle 1,?\rangle}, b_{\langle ?,?\rangle}\}. \tag{23}$$

The first four values of $\mathbb{D}$ denote concrete banks of the bank register in binary representation, e.g., $b_{\langle 1,0\rangle}$ represents bank 2. The remaining values of $\mathbb{D}$ describe (partially) unknown states of the bank register, e.g., $b_{\langle ?,0\rangle}$ represents a state of the bank register for which only the second bit is known and $b_{\langle ?,?\rangle}$ denotes a state for which all bits of the bank register are unknown.

To model the bits of the bank register, discrete variables P_u and Q_u become bit vector variables. The correctness constraint of Eqn. (3) is extended to take unknown bits of the bank register into account, i.e.,

$$\forall (u, v) \in E : \forall 1 \leq i \leq k : (P_v[i] \neq ?) \Rightarrow (Q_u[i] = P_v[i]), \tag{24}$$

where $P_v[i]$ and $Q_u[i]$ represent the ith bits of the discrete variables P_v and Q_u, respectively. Similarly, we adopt the cost functions for bank-sensitive (cf. Eqn. (7)) and transparent basic blocks (cf. Eqn. (9))

$$n\text{-}cost_u(P_u) = \sum_{1 \leq i \leq k} \begin{cases} c_u, & \text{if } fb_u[i] \neq ? \wedge P_u[i] \neq fb_u[i] \\ 0, & \text{otherwise} \end{cases} \tag{25}$$

$$e\text{-}cost_u(Q_u) = \sum_{1 \leq i \leq k} \begin{cases} c_u, & \text{if } Q_u[i] \neq ? \wedge Q_u[i] \neq lb_u[i] \\ 0, & \text{otherwise} \end{cases} \tag{26}$$

$$t\text{-}cost_u(P_u, Q_u) = \sum_{1 \leq i \leq k} \begin{cases} c_u, & \text{if } Q_u[i] \neq ? \wedge P_u[i] \neq Q_u[i] \\ 0, & \text{otherwise,} \end{cases} \tag{27}$$

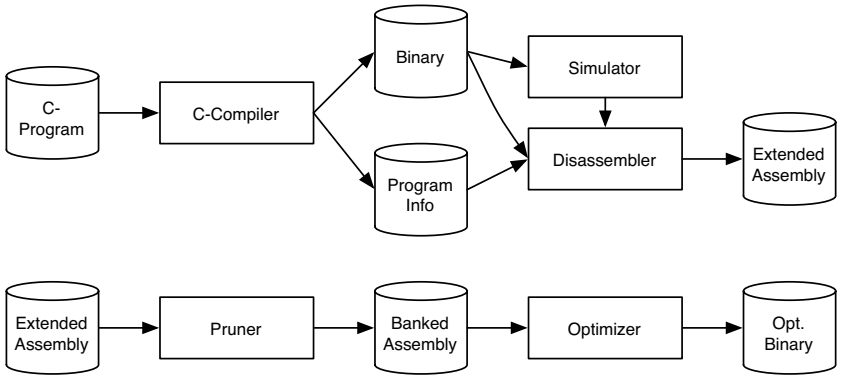


Fig. 8: Toolchain.

where for each bit that needs to be switched the costs c_u are imposed. The mapping of the non-uniform cost model to PBQP follows along the lines of the mapping devised in Section 4.3.

5. EXPERIMENTAL RESULTS

We evaluated our optimization using programs typically run on contemporary microcontrollers. Our sample included programs from the MiBench Embedded Benchmark Suite [Guthaus et al. 2001] and from DSPStone, and we surveyed a microcontroller real-time kernel and common microcontroller driver routines. We conducted this experiment on a PIC16F877A microcontroller [Microchip Technology Inc. 1997; 2003]. We applied different optimization objectives to show the versatility of our approach.

Our second experiment addressed the scalability of our optimization. This is important due to the prevailing trend to equip next generation microcontrollers with more program and data memory. (Microchip's PIC18F97J60 8-bit microcontroller for example provides 128KB of program memory and 4KB of data memory spread over 16 banks [Microchip Technology Inc. 2006].) In the second experiment we showed that our optimization

—scales well to a larger number of memory banks (i.e., 8, 16, 32),

—scales well to the larger problem sizes that will become feasible with future microcontrollers, and

—achieves the minimal result in almost all circumstances.

The second experiment involved the complete MiBench suite which has more than 6000 procedures. In this experiment we assumed the worst-case scenario for our optimization problem, which occurs when all basic blocks are transparent.

5.1 Experiment 1: PIC16F877A Microcontroller Benchmarks

The PIC family of midrange microcontrollers constitutes a RISC-based Harvard architecture with instruction sizes of 12, 14 or 16 bits, and a data-bus that is 8 bit wide [Microchip Technology Inc. 1997]. The PIC16F877A microcontroller provides

8KB of program memory and 368B of data memory spread over four banks [Microchip Technology Inc. 2003]. The MiBench programs that we could fit onto this microcontroller included "basicmath", "bitcount", "qsort", "sha", "CRC32" and "FFT". From the DSPStone benchmark suite we included "adpcm" and "matrix". We also surveyed a microcontroller real-time kernel and common microcontroller driver routines [MicrochipC.com 2006].

Our experimental setup is depicted in Figure 8. The compilation of a C benchmark program for the PIC16F877A resulted in a binary image and supplementary program information comprising the linker map file and a list of C-prototypes contained in the input program. We ran the binary image on the GPSIM simulator [Dattalo 2006] to obtain execution frequencies for the instructions of the binary image. The binary image together with the corresponding execution frequencies were then fed into the disassembler to produce an extended assembly file. The disassembler used the linker map file and the list of C-prototypes to establish procedural boundaries within the binary image. In this setup, an extended assembly file consisted of PIC assembly routines, where each instruction was augmented with its instruction frequency. The extended assembly file contained the bank selection instructions generated by the C-compiler. We used this information to compare it to the bank selection achieved by our optimization.

To carry out our optimization we pruned the extended assembly files from the bank selection instructions of the C-compiler. The pruner then performed data-flow analysis to annotate each operand of the assembly file with the required bank. In the following we call these annotations bank assertions, and the annotated assembly code is called banked assembly code.

The optimizer had to insert bank selection instructions into the banked assembly code so that all bank assertions were satisfied (this is the correctness criteria of our optimization) at minimal cost. *It should be noted that due to the pruner the bank selection of our optimizer was completely independent of the bank selection of the C-compiler.*

We checked the correctness of the inserted bank selection instructions generated by the optimizer. This was achieved by means of data-flow analysis. We determined the costs induced by the bank selection instructions in the optimized binary and compared them to the bank selection achieved by the C-compiler (from the extended assembly file). We used the HI-TECH PICC C-compiler as a reference point in this experiment. HI-TECH PICC is a high-performance C compiler advertised by Microchip itself for their whole family of PIC microcontrollers. It employs an optimizer that makes full use of PIC-specific features [HI-TECH Software 2006]. However, this compiler does not automatically assign C variables to memory banks (apart from the default assignment to bank 0). For this reason we had to manually assign program variables to memory banks with our benchmark programs. We replaced file I/O operations by I/O operations via the UART of the PIC16F877A. Some of the benchmark problem sizes had to be downsized to fit on an 8-bit microcontroller.

As depicted in Table III, we determined the overall static instruction count ("Total") together with the number of bank selection instructions ("BSL") for each benchmark. We determined the cycle counts induced by these instruction-catego-

Table III: Experimental results for the PIC microcontroller benchmark programs; program size reductions and speedups wrt. the optimizing commercial compiler.

		Instruction count		Cycle count		Size reduction		Speedup	
Benchmark	Objective	Total	BSL	Total	BSL	Total%	BSL%	Total%	BSL%
adpcm	HiT	6031	797	3.0e+09	3.9e+08	n/a	n/a	n/a	n/a
	Speed	5781	521	2.7e+09	1.6e+08	4.1	34.6	8.5	148.7
	Space	5640	406	2.8e+09	1.6e+08	6.5	49.1	8.3	140.2
	Mixed	5666	426	2.8e+09	1.6e+08	6.1	46.5	8.4	141.7
basicmath	HiT	1844	248	5856	1023	n/a	n/a	n/a	n/a
	Speed	1857	237	5545	712	-0.7	4.4	5.6	43.7
	Space	1794	198	5608	775	2.7	20.2	4.4	32.0
	Mixed	1799	199	5590	757	2.4	19.8	4.8	35.1
bitcnts	HiT	1888	197	81012	8091	n/a	n/a	n/a	n/a
	Speed	1871	163	75852	2931	0.9	17.3	6.8	176.0
	Space	1810	119	75992	3071	4.1	39.6	6.6	163.5
	Mixed	1811	120	75852	2931	4.1	39.1	6.8	176.0
crc32	HiT	450	36	2.2e+07	1.2e+06	n/a	n/a	n/a	n/a
	Speed	422	8	2.1e+07	64	6.2	77.8	5.9	1.9e+06
	Space	422	8	2.1e+07	64	6.2	77.8	5.9	1.9e+06
	Mixed	422	8	2.1e+07	64	6.2	77.8	5.9	1.9e+06
decimal	HiT	514	85	8547	1660	n/a	n/a	n/a	n/a
	Speed	460	31	7268	381	10.5	63.5	17.6	335.7
	Space	460	31	7284	397	10.5	63.5	17.3	318.1
	Mixed	460	31	7268	381	10.5	63.5	17.6	335.7
FFT	HiT	2693	348	53141	6485	n/a	n/a	n/a	n/a
	Speed	2576	216	50588	3932	4.3	37.9	5.0	64.9
	Space	2538	193	50688	4032	5.8	44.5	4.8	60.8
	Mixed	2540	195	50588	3932	5.7	44.0	5.0	64.9
lcd	HiT	307	65	72707	16285	n/a	n/a	n/a	n/a
	Speed	251	9	56431	9	18.2	86.2	28.8	1.8e+05
	Space	251	9	56431	9	18.2	86.2	28.8	1.8e+05
	Mixed	251	9	56431	9	18.2	86.2	28.8	1.8e+05
matrix	HiT	401	62	716	105	n/a	n/a	n/a	n/a
	Speed	368	27	660	49	8.2	56.5	8.5	114.3
	Space	360	21	667	56	10.2	66.1	7.3	87.5
	Mixed	360	21	666	55	10.2	66.1	7.5	90.9
nvmtsens	HiT	1238	134	121070	24229	n/a	n/a	n/a	n/a
	Speed	1138	34	97389	548	8.1	74.6	24.3	4.3e+03
	Space	1125	21	98075	1234	9.1	84.3	23.4	1.9e+03
	Mixed	1138	34	97389	548	8.1	74.6	24.3	4.3e+03
qsort	HiT	908	128	24549	3298	n/a	n/a	n/a	n/a
	Speed	802	20	21769	518	11.7	84.4	12.8	536.7
	Space	799	19	21771	520	12.0	85.2	12.8	534.2
	Mixed	799	19	21769	518	12.0	85.2	12.8	536.7
rtkernel	HiT	2097	422	2.6e+08	4.9e+07	n/a	n/a	n/a	n/a
	Speed	2063	343	2.3e+08	1.8e+07	1.6	18.7	13.4	169.9
	Space	1862	187	2.3e+08	1.8e+07	11.2	55.7	13.4	169.8
	Mixed	1864	189	2.3e+08	1.8e+07	11.1	55.2	13.4	169.9
serial	HiT	1178	189	8.4e+06	1.7e+06	n/a	n/a	n/a	n/a
	Speed	1035	44	6.7e+06	3128	12.1	76.7	25.3	5.4e+04
	Space	1022	33	6.7e+06	3385	13.2	82.5	25.3	5.0e+04
	Mixed	1022	33	6.7e+06	3385	13.2	82.5	25.3	5.0e+04
sha	HiT	3012	170	5.0e+06	9.3e+05	n/a	n/a	n/a	n/a
	Speed	2881	35	4.6e+06	5.2e+05	4.3	79.4	8.9	78.1
	Space	2868	26	4.7e+06	6.2e+05	4.8	84.7	6.8	51.8
	Mixed	2875	29	4.7e+06	5.8e+05	4.5	82.9	7.5	59.7
swi2c	HiT	740	123	4.1e+06	9.0e+05	n/a	n/a	n/a	n/a
	Speed	659	42	3.2e+06	673	10.9	65.9	28.1	1.3e+05
	Space	644	27	3.2e+06	1043	13.0	78.0	28.1	8.6e+04
	Mixed	653	36	3.2e+06	673	11.8	70.7	28.1	1.3e+05

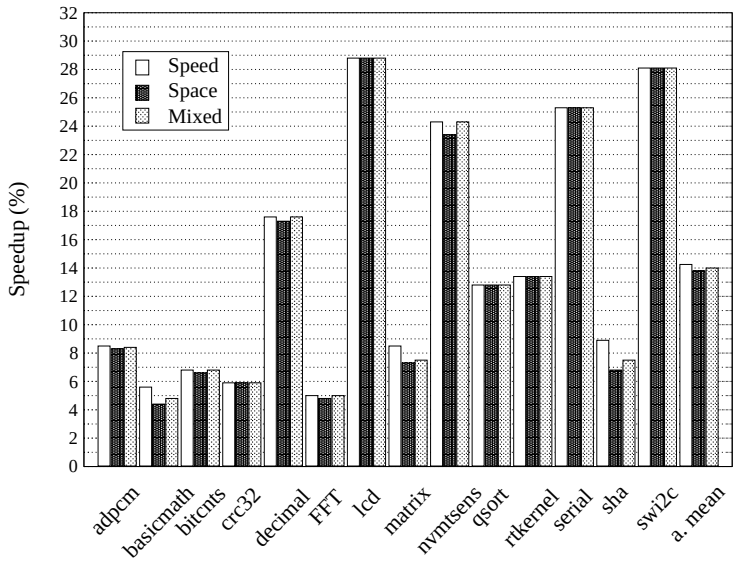


Fig. 9: Speedup achieved over the optimizing commercial compiler.

ries for a given benchmark sample input. The given "Total" cycle counts take into account extra jump instructions that might arise due to the insertion of bank selection instructions. For each benchmark these performance figures were determined for the HI-TECH PICC C-compiler and for our optimization. We performed our optimization under the objectives (1) speed, (2) space, and (3) mixed (a combination of speed and space). The corresponding values for α and β were $\alpha = 1$, $\beta = 0$ (speed), $\alpha = 0$, $\beta = 1$ (space), and $\alpha = 0.5$, $\beta = 0.5$ (mixed). In terms of Eqns. (11) and (12), we set bsl-cycles = bsl-size = jump-cycles = jump-size = 1. The rightmost columns in Table III depict the memory footprint reduction and the resulting performance improvement achieved by our optimization. The figures reflect the goals of the different optimization objectives: optimizing for space results in the lowest number of issued bank selection instructions, whereas optimizing for speed minimizes instruction cycles. Optimizing for speed and space combines both objectives, resulting in performance figures between the two. Note however, that for some benchmarks the speed and space optimizations are identical, which then applies for the mixed optimization as well.

It follows from Table III that the reduction of the program memory footprint (corresponding to the overall instruction count) is between 2.7% and 18.2% when we optimize for *space*. In this case the reduction of bank selection instructions is between 20.2% and 86.2%. If we optimize for *speed*, the achieved overall improvement is between 5.0% and 28.8%, and the improvement with respect to the execution of bank selection instructions alone is between 43.7% and 1900000%. Our optimization achieved the optimal solution for all benchmark programs. The overall speedup is shown in the bar chart of Figure 9, and the program-size reduction of our bank selection optimization is shown in Figure 10 ("a. mean" denotes the arithmetic mean over all benchmark programs).

To investigate the sensitivity of our optimization to actual input data, we em-

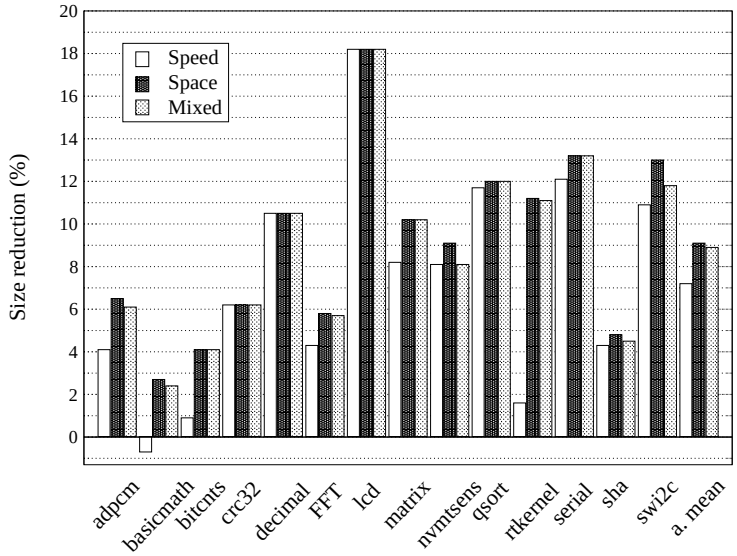


Fig. 10: Program size reduction achieved over the optimizing commercial compiler.

Table IV: Optimization results across different data sets.

	Average speedup(%)						Average size reduction(%)					
Benchm.	Speed	Space	Mixed	σ_{Spd}	σ_{Spc}	σ_{Mxd}	Speed	Space	Mixed	σ_{Spd}	σ_{Spc}	σ_{Mxd}
adpcm	8.38	8.17	8.21	0.18	0.19	0.17	3.88	6.5	6.05	0.33	0	0.05
bitcnts	6.86	6.67	6.86	0.09	0.08	0.09	1.35	4.1	4.1	0.7	0	0
crc32	5.9			0			6.2			0		
FFT	5.04	4.83	5.04	0.05			4.3	5.8	5.7	0		
qsort	12.54	12.53	12.54	0.26			11.45	12.0	11.98	0.36	0	0.08
sha	8.9	6.8	7.5	0			4.3	4.8	4.5	0		

ployed the MiDataSets of Fursin et al. [2007]. The MiDataSets provide 20 data sets per MiBench benchmark, with the intention to establish a baseline for the evaluation of compiler optimizations under varying input data conditions. We profiled several of our benchmarks with each of the corresponding 20 data sets to determine basic block execution frequencies. The MiDataSets for qsort had to be shrunk to fit onto the PIC16F877A microcontroller. For benchmark FFT we used the data from bitcnts to seed the pseudo-random generator. Table IV shows the average speedup and program size reduction together with the standard deviation σ across all data sets. Note that a standard deviation of zero from the average footprint size reduction is expected when optimizing for space, because execution frequencies are not taken into account in this case. The extremely small values for σ in Table IV show that our optimizations were effective with each data set.

We investigated the instruction distributions of the PIC microcontroller benchmark programs to determine applicability and improvement potential of our optimization. The obtained data is depicted in Table V. Therein Column "traB" denotes the percentage of transparent basic blocks. "traS", "isenS" and "bsenS" denote the percentages of transparent statements, bank sensitive statements inside basic blocks, and first and last bank sensitive statements of a basic block (note that they add up to 100% and hence do not account for `BSL` statements). "traC", "isenC"

Table V: Instruction distribution properties of the PIC microcontroller benchmark programs.

Benchmark	Objective	traB	traS	isenS	bsenS	traC	isenC	bsenC	bBslS	bBslC	bsl
adpcm	HiT	63.4	76.6	7.8	15.6	77.8	7.4	14.8	71.9	77.7	13.2
	Speed	65.7	77.9	7.4	14.7	77.8	7.4	14.8	60.1	47.8	9.0
	Space	65.6	78.0	7.4	14.6	77.8	7.4	14.8	49.3	49.6	7.2
basicmath	HiT	36.6	73.4	16.6	10.0	68.7	17.3	14.0	37.5	51.7	13.4
	Speed	39.1	72.4	16.8	10.8	68.1	17.4	14.5	40.5	37.2	12.8
	Space	33.7	72.7	16.7	10.6	68.3	17.3	14.4	28.8	42.3	11.0
bitcnts	HiT	50.9	70.2	14.3	15.5	68.9	14.0	17.1	61.9	77.8	10.4
	Speed	53.5	71.3	13.8	14.9	68.9	14.0	17.1	54.6	44.3	8.7
	Space	51.9	71.0	14.0	15.0	68.9	14.0	17.1	37.8	46.8	6.6
crc32	HiT	50.8	73.9	15.2	10.9	77.7	7.4	14.9	94.4	100.0	8.0
	Speed	60.7	79.2	12.3	8.5	77.7	7.4	14.9	100.0	100.0	1.9
	Space	60.7	79.2	12.3	8.5	77.7	7.4	14.9	100.0	100.0	1.9
decimal	HiT	56.3	68.8	13.3	17.9	77.2	8.0	14.8	85.9	86.4	16.5
	Speed	56.3	68.8	13.3	17.9	77.2	8.0	14.8	61.3	40.7	6.7
	Space	56.3	68.8	13.3	17.9	77.2	8.0	14.8	61.3	43.1	6.7
FFT	HiT	46.4	72.9	17.1	10.0	71.9	16.2	11.9	57.2	63.7	12.9
	Speed	48.9	72.8	17.3	9.9	71.9	16.2	11.9	37.0	44.5	8.4
	Space	47.1	73.0	17.2	9.8	71.9	16.2	11.9	31.6	46.1	7.6
lcd	HiT	56.0	67.7	8.7	23.6	84.5	0.4	15.1	92.3	100.0	21.2
	Speed	56.0	67.7	8.7	23.6	84.5	0.4	15.1	44.4	44.4	3.6
	Space	56.0	67.7	8.7	23.6	84.5	0.4	15.1	44.4	44.4	3.6
matrix	HiT	80.0	90.3	5.3	4.4	86.4	7.7	5.9	69.4	62.9	15.5
	Speed	80.7	90.3	5.3	4.4	86.4	7.7	5.9	40.7	20.4	7.3
	Space	80.0	90.9	4.7	4.4	86.4	7.7	5.9	33.3	30.4	5.8
nvmtsens	HiT	50.1	75.7	0.9	23.4	73.0	0.9	26.1	89.6	99.0	10.8
	Speed	50.3	75.8	0.9	23.3	73.0	0.9	26.1	58.8	56.4	3.0
	Space	50.3	75.8	0.9	23.3	73.0	0.9	26.1	33.3	80.6	1.9
qsort	HiT	55.8	69.5	10.0	20.5	68.8	9.0	22.2	85.9	97.4	14.1
	Speed	56.1	69.5	10.0	20.5	68.8	9.0	22.2	45.0	89.6	2.5
	Space	55.8	69.5	10.0	20.5	68.8	9.0	22.2	42.1	89.6	2.4
rtkernel	HiT	56.1	68.2	8.7	23.1	78.7	3.2	18.1	71.6	99.9	20.1
	Speed	60.6	71.3	7.2	21.5	78.7	3.2	18.1	72.9	99.6	16.6
	Space	60.9	71.9	7.3	20.8	78.7	3.2	18.1	50.3	99.6	10.0
serial	HiT	53.8	70.9	6.6	22.5	77.0	0.1	22.9	88.4	99.9	16.0
	Speed	56.3	73.0	5.7	21.3	77.0	0.1	22.9	50.0	25.1	4.3
	Space	56.0	73.6	5.1	21.3	77.0	0.1	22.9	36.4	30.8	3.2
sha	HiT	46.9	66.5	20.1	13.4	60.9	17.7	21.4	94.7	70.2	5.6
	Speed	47.3	66.5	20.1	13.4	60.9	17.7	21.4	74.3	47.0	1.2
	Space	46.9	66.5	20.1	13.4	60.9	17.7	21.4	65.4	54.8	0.9
swi2c	HiT	52.9	72.9	2.1	25.0	84.8	0.2	15.0	95.1	100.0	16.6
	Speed	52.9	72.9	2.1	25.0	84.8	0.2	15.0	85.7	86.3	6.4
	Space	52.9	72.9	2.1	25.0	84.8	0.2	15.0	77.8	91.2	4.2

and "bsenC" denote the corresponding percentages of the cycle counts. Given the total number of `BSL` statements, "bBslS" denotes the percentage of `BSL` statements before the first and after the last bank-sensitive statement of a basic block, and "bBslC" denotes the corresponding cycle count percentage. Given the total number of statements of a benchmark, "bsl" denotes the percentage of `BSL` statements. For the following considerations we use again HI-TECH PICC as a baseline for comparison. However, as already pointed out, our optimization is independent of this baseline. In Column 2 of Table V, "HiT" denotes data obtained from programs compiled with the HI-TECH compiler, and "Speed" and "Space" denote the corresponding objectives of our optimization.

The number of `BSL` statements in the code is an upper bound for the number of

statements that can be removed by our optimization. Column "bsl" shows that the percentages of `BSL` statements for "HiT" range from 5.6% to 21.2%, which indicates potential for improvement. However, due to the non-uniform cost model of the PIC16F877A (cf. Section 4.5), the number of optimization opportunities is less than the actual number of `BSL` statements. In our survey of benchmark programs, 49% of all bank switches require two `BSL` statements (as depicted in Eqn. (21)). This suggests an improvement potential for the partitioning of variables. It is however the case that most of the special function registers of the PIC 16Fxxx architecture are spread across different banks, which limits data partitioning optimizations for hardware-dependent code.

Transparent basic blocks benefit our optimization because they constitute potential insertion points for `BSL` statements. Column "traB" shows that between 36.6% and 80% of all basic blocks in the "HiT" code are transparent. It should be noted that due to the splitting of critical edges our optimization may generate additional transparent basic blocks and associated statements; for this reason the percentages given for basic blocks and statements may vary across different optimizations of a given benchmark.

Our optimization is effective for first and last bank-sensitive statements of a basic block. Column "bsenS" of Table V shows that between 4.4% and 25% of all bank-sensitive statements are either a first or last bank sensitive statement in a basic block. For the majority of benchmark programs the number of bank-sensitive statements in the above category dominates the number of bank-sensitive statements between the first and last bank-sensitive statement ("isenS"). Figure 10 confirms that our space optimization is most effective for benchmarks in the high (17.9%–25%) "bsenS" category. "Matrix" from the DSPStone benchmark suite is the only exception to this rule. Despite only 4.4% of bank-sensitive statements in the "bsenS" category, our optimization achieves a program size reduction of 10.2%. This can be attributed to the exceptionally high rate of transparent basic blocks (80%) and to the high improvement potential (15.5% share of `BSL` statements) present in the "HiT" code that we use for comparison. In absolute terms "bsl" is down to 5.8% after our space optimization, which is is well within the range of 0.9%–11% that we achieve when optimizing for space.

The principles outlined above apply to cycle counts and optimizations in the time-domain as well. However, one has to take into account the non-uniformity of instruction frequencies among different statements to be able to precisely relate benchmark programs to the achieved speedups. All surveyed benchmarks exhibit a surprisingly high percentage of transparent statements (between 66.5% and 90.3%). This suggests that there is a potential for instruction reordering within basic blocks to further improve the bank selection optimization.

We compared the efficiency and effectiveness of a heuristic PBQP solver with a branch-and-bound PBQP solver. For the experiment we used the inter-procedural space optimization only. Because of the small number of RN nodes, the heuristic solver guessed the optimal solution and the branch-and-bound solver was able to terminate the search quickly. The results of the experiment are shown in Table VI. The first two columns ("nodes" and "edges") show the number of discrete variables and the number of dependencies between two discrete variables for each benchmark.

Table VI: PBQP Problem, Interprocedural Optimization for Space.

	PBQP		Heuristic							B&B			
Benchmark	nodes	edges	R0	RI	RII	RN	mem	t	f	#	mem	t	f
adpcm	2888	2862	297	1812	775	4	3.8	0.19	134	6	3.86	0.19	134
basicmath	344	293	75	244	22	3	0.37	0.02	33	5	0.44	0.02	33
bitcnts	754	689	132	452	170	0	0.9	0.04	23	1	0.9	0.05	23
crc32	106	81	30	66	10	0	0.1	0.01	4	1	0.1	0.01	4
decimal	238	219	25	195	18	0	0.27	0.02	12	1	0.27	0.02	12
FFT	566	492	87	428	51	0	0.61	0.03	37	1	0.61	0.03	37
lcd	200	196	11	169	20	0	0.24	0.01	2	1	0.24	0.01	2
matrix	110	100	18	80	12	0	0.12	0.01	3	1	0.12	0.01	3
nvmtsens	946	1108	16	491	436	3	1.61	0.09	6	4	1.79	0.11	6
qsort	394	357	51	288	55	0	0.45	0.02	6	1	0.45	0.02	6
rtkernel	1252	1041	241	937	72	2	1.29	0.04	45	3	1.32	0.05	45
serial	714	677	76	573	63	2	0.82	0.04	8	4	0.87	0.04	8
sha	1020	958	134	736	150	0	1.19	0.05	10	1	1.19	0.06	10
swi2c	510	506	24	401	83	2	0.65	0.03	20	3	0.69	0.03	20

The majority of the benchmarks result in small PBQP problems of approx. 1000 variables or less for the bank selection optimization. There is only one benchmark ("adpcm") that has approx. 3000 discrete variables.

The small number of dependencies ("edges") is already a good indication that the number of RN nodes is small. The heuristic solver can reduce nearly all nodes with reductions R0, RI, and RII. Six benchmarks have RN nodes, for which the heuristic solver cannot guarantee optimality[4]. The number of R0, RI, RII, and RN nodes are given in Table VI. The values of the objective functions are shown under Columns f. The heuristic solutions coincide with the branch-and-bound solutions. Note that Column "#" gives the total number of sub-problems that were computed for the branch and bound solver. For benchmarks that do not contain RN nodes, this number is equal to one. Due to a very tight bound [Hames and Scholz 2006], the branch-and-bound solver of PBQP finds the optimal solution with only a few sub-problems. Benchmark "adpcm" required 6 sub-problems to compute the optimal solution whereas all the other benchmarks with RN nodes needed less than 6 sub-problems[5]. As shown in the table, the heuristic PBQP solver delivers an optimal solution even for benchmarks with RN nodes, i.e. the Columns "f" of the solvers coincide for all benchmarks.

The memory consumption in MBytes is listed in Column "mem" and the time measurements in Column "t". Time measurements have a time resolution of 10ms and the solvetime is given in seconds. Both solvers were executed on a Pentium 4 with 1.8Ghz and 1.2GB RAM under Linux. The memory consumption and the execution time of the branch-and-bound solver is marginally larger than the heuristic solver because of storing and traversing the branch and bound tree. Runtime and memory consumption are negligible for both solvers.

[4]A heuristic is only applied for RN nodes. If there are no RN nodes, the heuristic solver obtains an optimal solution.

[5]For benchmarks with no RN nodes, the solution is optimal and the total number of sub-problems is one, i.e., the problem itself.

Table VII: Problem sizes and PBQP optimization results for MiBench.

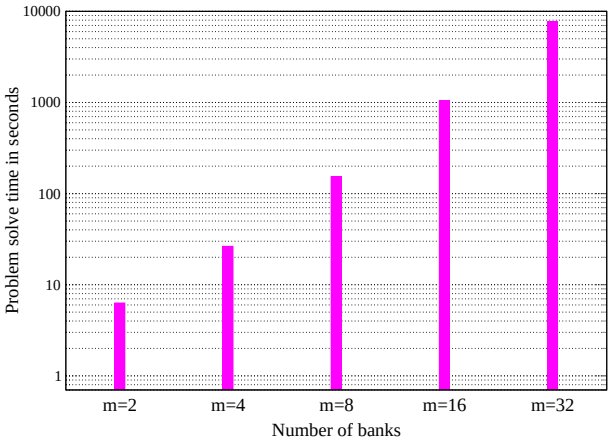
		CFG			CFG (simpl.)		PBQP		
	Benchmark	proc	node	edge	node	edge	P,Q	optproc	
autom.	basicmath	6	97	143	174	220	348	6	100.0%
	bitcount	15	88	102	124	138	248	13	86.7%
	qsort	4	40	55	59	74	118	3	75.0%
	susan	19	698	1102	1104	1509	2208	11	57.9%
consum.	jpeg	375	6998	10255	10948	14211	21896	262	69.9%
	lame	213	5720	8510	8735	11528	17470	148	69.5%
	mad	1	3	2	3	2	6	1	100.0%
	tiff	510	10735	15668	15678	20628	31356	320	62.7%
	typeset	399	20650	31675	31322	42348	62644	251	62.9%
nw	dijkstra	12	138	184	198	246	396	10	83.3%
	patricia	5	160	227	217	285	434	2	40.0%
office	ghostscript	3551	47967	65967	67117	85120	134234	2734	77.0%
	ispell	107	2368	3611	3757	5001	7514	51	47.7%
	rsynth	53	1326	2012	2074	2760	4148	32	60.4%
	sphinx	684	10597	14895	15732	20043	31464	522	76.3%
	stringsearch	13	176	240	262	326	524	7	53.8%
secur.	blowfish	14	237	344	373	483	746	13	92.9%
	pgp	320	7381	10959	11207	14788	22414	195	60.9%
	rijndael	7	157	223	220	286	440	2	28.6%
	sha	7	56	70	78	92	156	7	100.0%
telec.	adpcm	5	79	105	103	132	206	3	60.0%
	CRC32	4	32	41	47	56	94	3	75.0%
	FFT	6	83	114	124	156	248	6	100.0%
	gsm	65	1589	2244	2097	2754	4194	44	67.7%
	all	6395	117375	168748	171753	223186	343506	4646	72.7%

5.2 Experiment 2: Scalability Study

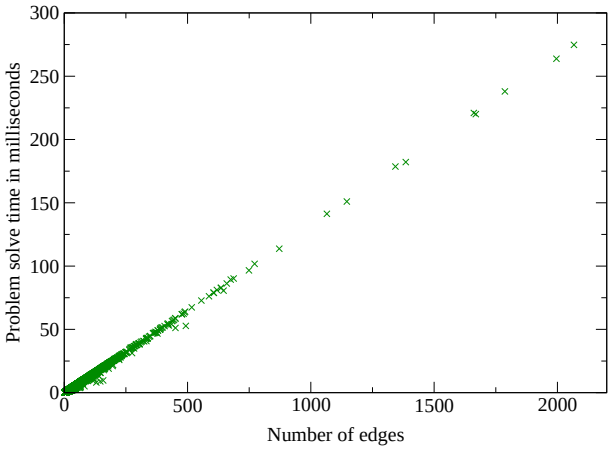
In our second experiment we investigated the scalability of our optimization to future microcontrollers which will provide a larger number of memory banks and larger program memories. Due to the memory constraints of current microcontrollers it was not possible to investigate large MiBench programs within Experiment 1. Instead we used the CFGs and constructed PBQP problems based on the control flow information. This approach is justified by the fact that the runtime of the PBQP problem solely depends on (1) the size and topology of the CFGs of the input programs, and (2) the number of banks. The following evaluation of our optimizations with MiBench was done on a Linux system based on kernel version 2.6 and on GCC version 3.2.3.

During a build of the complete MiBench source distribution we used the information provided by the -dw option of GCC to generate CFGs for all MiBench functions. With these CFGs we assumed that all basic blocks are transparent. As pointed out in Section 4.2, this assumption constitutes the worst-case for our PBQP optimization problem because it requires optimizations across basic blocks (cf. Eqn. (14)). This experiment was conducted to gather empirical data on (1) the discrete optimization problem and the PBQP solver (esp. on the optimality of the derived solution), and (2) the execution time spent for the analysis.

Table VII enumerates the programs of the MiBench suite together with the number of functions and CFG nodes and edges. It lists the number of nodes and edges of the so-called *simplified* CFG which is derived from the input-CFG by remov-



(a) Accumulated solve times wrt. the number of banks.



(b) Solve times of functions wrt. problem size.

Fig. 11: MiBench solve times.

ing unreachable nodes and splitting critical edges[6]. Our analysis is based solely on simplified CFGs. It should be noted that these CFGs correspond to GCC's intermediate representation (RTL), which is on a higher abstraction level than the CFGs created by the HI-TECH PICC compiler. Regarding the discrete optimization problem itself, we list the number of discrete variables occurring with our approach ("P,Q"). Column "optproc" depicts the number of functions and their percentages in terms of the overall numbers of functions for which our PBQP solver could derive an optimal solution[7]. Since the PBQP problem is NP complete in the general case, this figure is an important indication of the practicality of our bank

[6]Section 4.2 shows that splitting a critical edge creates a critical block, which accounts for the actual increase in the number of nodes (cf. also Figure 2(a) and Figure 2(b)).
[7]It follows from Section 2 that, if our solver derives an optimal solution, it does not need to apply an RN reduction.

selection optimization. It follows that despite the assumption that all basic blocks are transparent (which is a highly unrealistic worst-case scenario for practical programs, as indicated in Table V), we can compute the minimal solution for more than 72% of the 6395 MiBench functions.

Our PBQP optimization problem has complexity $\mathcal{O}(n \cdot m^3)$, where n is the number of discrete variables and m is the number of banks. Since m is fixed for any practical implementation, we can expect our optimization to run in almost linear time. We have conducted measurements using the k-best measurement scheme described by Bryant and O'Halloran [2003] and achieved an epsilon of $\varepsilon = 25\%$ by selecting the 5 best measurements out of 50 measurements. Figure 11(a) shows the problem solve times measured for m=2,4,8,16, and 32 banks on a logarithmic scale. For m=32 banks the PBQP problem takes 7700 seconds on a 1.8GHz Pentium 4 PC to solve the optimizations for the *entire* MiBench suite. On average, that is 1.2 seconds per function, which suggests that the overhead induced by our optimization is acceptable for practical implementations. Figure 11(b) details the problem solve times of functions for m=4 banks with respect to the number of CFG edges.

6. CONCLUSION

We believe this is the first algorithmic approach to address the problem of minimizing the number of bank selection instructions for a given instruction order and a given data partitioning. We formulated the bank selection placement problem as a discrete optimization problem. Optimization objectives such as speed, space or energy are modeled as cost metrics and allow parameterizable trade-offs between them. We provided empirical evidence that our method performs well for embedded systems architectures. We devised an efficient algorithm for bank selection minimization based on Partitioned Boolean Quadratic Programming.

We conducted experiments with programs from the DSPStone and MiBench benchmark suites, and we surveyed a microcontroller real-time kernel and common microcontroller driver routines. To show the practicality of our approach, we implemented a toolchain for the Microchip PIC 16F877A microcontroller. This toolchain optimizes the bank selections of binaries. For the surveyed programs we achieved speedups to 28.8% and code size reductions up to 18.2%, where the base-line is a state-of-the-art C-compiler for the PIC 16F877A.

In this work we showed that nearly optimal bank selection is solvable in polynomial time. We conducted experiments which show that the introduced optimization technique has low overhead and performs very well on a real-world benchmark. Also a worst-case scenario (assuming that all basic blocks are transparent) computes insertions of bank selections in almost linear time.

ACKNOWLEDGMENTS

We would like to thank Merrilee Robb and Wei-Ying Ho for proofreading the manuscript. We would like to thank Sanjay Chawla for presenting the paper [Kleinberg and Tardos 1999] in our algorithmic reading group.

REFERENCES

Banakar, R., Steinke, S., Lee, B., Balakrishnan, M., and Marwedel, P. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of*

the 10th International Symposium on Hardware/Software Codesign (CODES'02). ACM Press, New York, NY, USA, 73–78.

Bryant, R. E. and O'Halloran, D. R. 2003. *Computer Systems: A Programmer's Perspective*. Prentice-Hall.

Cai, Q. and Xue, J. 2003. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 91–102.

Cho, J., Paek, Y., and Whalley, D. 2004. Fast memory bank assignment for fixed-point digital signal processors. *ACM Transactions on Design Automation of Electronic Systems 9,* 1, 52–74.

Dattalo, T. S. 2006. The Gpsim SW simulator for PIC microcontrollers. `http://www.dattalo.com/gnupic/gpsim.html`.

Delaluz, V., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. 2000. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00)*. ACM Press, New York, NY, USA, 138–147.

Eckstein, E. 2003. Code optimizations for digital signal processors. Ph.D. thesis, Institute of Computer Languages, Compilers and Languages Group, Vienna University of Technology.

Fursin, G., Cavazos, J., O'Boyle, M., and Temam, O. 2007. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*. Vol. 4367. Springer LNCS, 245–260.

Gartner Dataquest. 2004. 2003 microcontroller market share and unit shipments.

Gartner Dataquest. 2005. Top companies revenue from shipments of 8-bit mcu — all applications.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. IEEE Computer Society, 3–14.

Hames, L. and Scholz, B. 2006. Nearly optimal register allocation with PBQP. In *Proceedings of the 7th Joint Modular Languages Conference (JMLC'06)*. LNCS, vol. 4228. Springer, 346–361.

Hempstead, M., Tripathi, N., Mauro, P., Wei, G.-Y., and Brooks, D. 2005. An ultra low power system architecture for sensor network applications. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Washington, DC, USA, 208–219.

Hempstead, M., Wei, G., and Brooks, D. 2006. Architecture and circuit techniques for low-throughput, energy-constrained systems across technology generations. In *Proceedings of the 2006 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'06)*. ACM Press, 368–378.

HI-TECH Software. 2006. PICC ANSI C Compiler. `http://www.htsoft.com/`.

Kiyohara, T., Mahlke, S., Chen, W., Bringmann, R., Hank, R., Anik, S., and Hwu, W.-M. 1993. Register connection: A new approach to adding registers into instruction set architectures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 247–256.

Kleinberg, J. M. and Tardos, E. 1999. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and markov random fields. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. IEEE Computer Society, Washington, DC, USA, 14–23.

Knoop, J., Rüthing, O., and Steffen, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16,* 4, 1117–1155.

Leupers, R. and Kotte, D. 2001. Variable partitioning for dual memory bank DSPs. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. 1121–1124.

Li, L., Gao, L., and Xue, J. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*. 329–338.

Microchip Technology Inc. 1997. PICmicro mid-range MCU family reference manual.

Microchip Technology Inc. 2003. PIC16F87XA data sheet.

Microchip Technology Inc. 2006. PIC18F97J60 family data sheet, advance information.

MicrochipC.com. 2006. PIC micros and C. `http://www.microchipc.com/`.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nazhandali, L., Minuth, M., Zhai, B., Olson, J., Austin, T., and Blaauw, D. 2005. A second-generation sensor network processor with application-driven memory optimizations and out-of-order execution. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. ACM Press, New York, NY, USA, 249–256.

Nystrom, E. and Eichenberger, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 103–114.

Panda, P. R., Catthoor, F., Dutt, N. D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., and Kjeldsberg, P. G. 2001. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems 6,* 2, 149–206.

Panda, P. R., Dutt, N. D., and Nicolau, A. 2000. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems 5,* 3, 682–704.

Ravindran, R. A., Senger, R. M., Marsman, E. D., Dasika, G. S., Guthaus, M. R., Mahlke, S. A., and Brown, R. B. 2005. Partitioning variables across register windows to reduce spill code in a low-power processor. *IEEE Trans. Comput. 54,* 8, 998–1012.

Saghir, M. A. R., Chow, P., and Lee, C. G. 1996. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York, NY, USA, 234–243.

Scholz, B. and Eckstein, E. 2002. Register allocation for irregular architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'02)*. ACM, 139–148.

Scholz, B., Horspool, N., and Knoop, J. 2004. Optimizing for space and time usage with speculative partial redundancy elimination. *SIGPLAN Notices 39,* 7, 221–230.

Sudarsanam, A. and Malik, S. 1995. Memory bank and register allocation in software synthesis for ASIPs. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*. 388–392.

Udayakumaran, S. and Barua, R. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. ACM Press, 276–286.

Verma, M., Wehmeyer, L., and Marwedel, P. 2004. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*. IEEE Computer Society, Washington, DC, USA, 1264–1269.

Zhuang, X., Pande, S., and Jr., J. S. G. 2002. A framework for parallelizing load/stores on embedded processors. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE Computer Society, 68–79.

Zhuge, Q., Xiao, B., and Sha, E. H.-M. 2002. Variable partitioning and scheduling of multiple memory architectures for DSP. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society, Washington, DC, USA, 332.

Received Month Year; revised Month Year; accepted Month Year