Minimizing Bank Selection Instructions for Partitioned Memory Architectures*

Bernhard Scholz
The University of Sydney
scholz@cs.usyd.edu.au

Bernd Burgstaller
The University of Sydney
bburg@cs.usyd.edu.au

Jingling Xue
University of NSW
jxue@cse.unsw.edu.au

ABSTRACT

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 optimal placement is controlled by a variety of different objectives, such as runtime, low power, small code size or a combination of these parameters. We have formulated the problem as a form of Partitioned Boolean Quadratic Programming (PBQP).

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 of program memory space between 2.7% and 18.2%, and an overall improvement with respect to instruction cycles between 5.1% and 28.8%. Our optimization achieved an optimal solution for all benchmark programs.

Categories and Subject Descriptors

D3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Languages, Performance

Keywords

Compiler optimization, microcontrollers, partitioned memory architecture, bank-switching, RAM allocation, PBQP

*This project has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190 and the University of Sydney R&D Grants Scheme "Speculative Partial Redundancy Elimination" under Contract L2849 U3229.

CASES'06, October 23–25, 2006, Seoul, Korea.

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. According to the Semiconductor Industry Association's November 2005 forecast, the market for 4-, 8-, 16-, and 32-bit microcontrollers will 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 [6].

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 mechanical-based systems.

Bank switching is a common technique for 8-bit microcontrollers to increase 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 addresses 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 PIC16F877A microcontroller allows data accesses to be switched between four 128B data banks. 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 [15].

The disadvantage of bank switched architectures is the code-size and runtime overhead caused by bank selection instructions. Several commercial and open-source compilers for microcontrollers provide limited support to generate bank-switched code. For example, GNU GCC for Motorola

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 of, the bank switching schemes used in existing compilers seem to be ad hoc, and it is still a challenging research problem to generate efficient memory accesses for bank-switched architectures.

This work is concerned with developing a compiler optimization for optimal 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 a minimum number of bank selection instructions in the program to guarantee that banked memory is accessed correctly. The optimal placement is controlled by a variety of objectives such as runtime, low power, small code size or a combination of these parameters. The authors are only aware of an ad-hoc approach in this area, which was introduced in [8].

Most previous efforts on partitioned memory architectures focus on maximizing parallel data accesses to make memory banks simultaneously active [2, 10, 16, 17, 19, 21, 25, 26]. 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 reorganize the order of instructions and the layout of data, e.g., by loop transformations [3], 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 [1, 11, 23, 24] and the references therein for a number of compiler techniques proposed on performing 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 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, 2004 [5]). 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 and a combination of both.

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* 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. 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 $\{v|(u,v) \in E\}$. A critical edge is an edge (u,v) for which $|succs(u)| > 1$ and $|preds(v)| > 1$. 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 [20, 4] 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

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

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

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 $\vec{x_i}^T \vec{1} = 1$ that restricts vectors $\vec{x_i}$ such that only one vector element is assigned one; all other elements are set to zero.

The related 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 as 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 $\vec{c_i}$. Quadratic forms and scalar products are employed to formulate PBQP as a mathematical program:

$$s.t.\ \forall 1 \leq i \leq n : \vec{x}_i \in \{0,1\}^{|\mathbb{D}_i|}$$
$$\forall 1 \leq i \leq n : \vec{x}_i^T \vec{1} = 1$$
$$\min f = \sum_{1 \leq i \leq n} \vec{x}_i^T \vec{c}_i + \sum_{1 \leq i < j \leq n} \vec{x}_i^T \mathcal{C}_{ij} \vec{x}_j.$$

In [20, 4] 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. $m = \max(|\mathbb{D}_1|, \ldots, |\mathbb{D}_n|)$. For a given problem, the solver eliminates discrete variables until the problem is trivially solvable. Each elimination step requires a reduction. The solver has reductions R0, RI, RII, which are not always applicable. If no reduction can be applied, the problem becomes irreducible and a heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable $\vec{x_i}$ and a good assignment for it by searching for local minima. The obtained solution is guaranteed to be optimal if the reduction RN is not used.

3. MOTIVATION

The goal of our optimization is to insert a *minimum* number of bank selection instructions while ensuring that banked memory is accessed correctly. The underlying optimization assumptions are that all variables in a program have been assigned to memory banks and that we do 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 in a 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 bank of the access is made active prior to its execution. Otherwise, the program semantics is 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 made active after returning from the function call. 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.

The local predicate $bank(s)$ gives the bank property of statement s. It is defined as follows:

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

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 an optimal 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 or not. 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 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 the transparent basic block.

Splitting critical edges creates a transparent basic block in the control flow graph (a so called *critical basic block*). Critical basic blocks are potential hosts for bank selection instructions and therefore yield optimization opportunities. However, splitting critical edges is not free because additional jump statements need to be inserted to split an edge. Note that the intra-procedural optimization only splits critical edges if a bank selection instruction is eventually inserted on the critical edge. Otherwise, it is only considered in the cost analysis of the optimization.

Consider our running example in Fig. 1(a), where each basic block is numbered with its basic block number shown to its left in **bold**. Fig. 1(b) shows the resulting CFG where all five critical edges have been split tentatively. The underlined numbers alongside CFG edges represent execution frequencies. The execution frequency of a basic block is the sum of frequencies of either its incoming or outgoing CFG edges. 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, respectively 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`.

Figs. 1(c)-(e) illustrate the optimal 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 Fig. 1(b)). We assume that bank selection instruction 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 optimal solution for speed is shown in Fig. 1(c) where we place

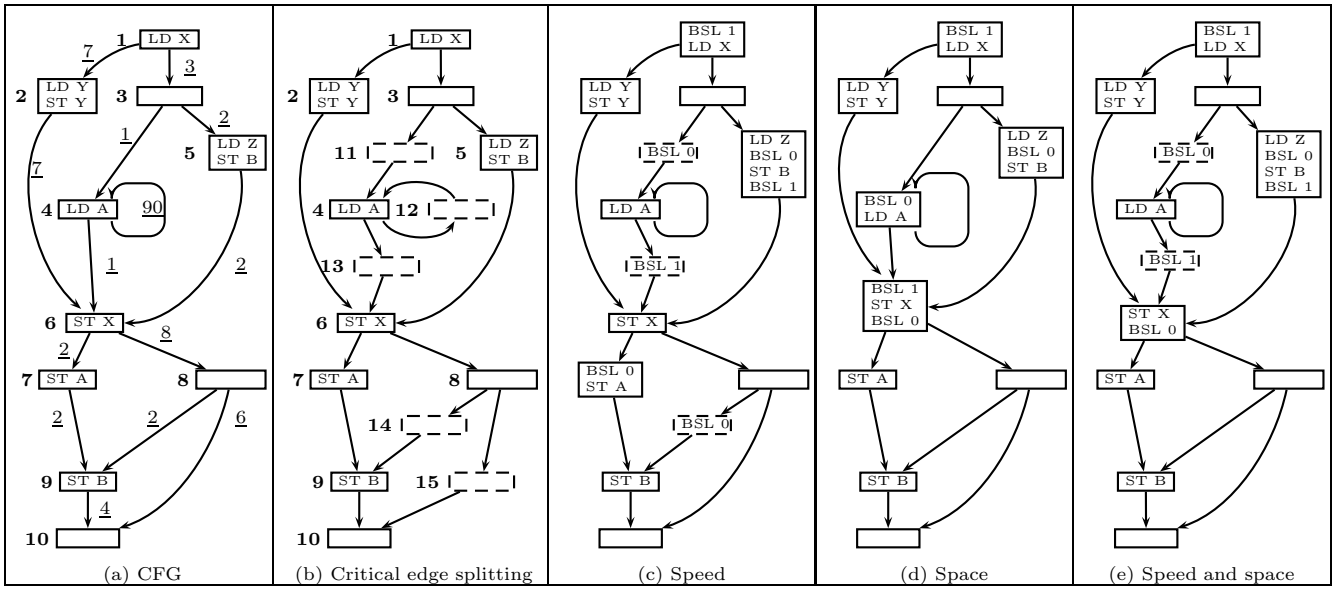

Figure 1: An example for a two-bank architecture (`A` and `B` reside in bank 0 and `X`, `Y` and `Z` in bank 1).

a bank selection instruction before and after the loop and in basic block 14. In Fig. 1(d) the optmisation 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 Fig. 1(e). For this optimization the speed objective is weighted a third and the space objective is weighted two thirds.

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 intra-procedural 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 intra- procedural 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 Fig. 2, 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, 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

```
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)
```

Figure 2: Local optimisation.

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 Fig. 2. We use fb_u to denote the bank of statement $first(u)$ and lb_u to denote the bank of statement $last(u)$.

Fig. 3 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 in-

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

Figure 3: Example (A and B reside in bank 0 and X,Y, and Z in bank 1).

troduce transparency to this example. The call to function `foo` in statement s_3 potentially modifies the bank select 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.

4.2 Intra-Procedural Optimization

The intra-procedural optimization is effective because bank selection instructions can be hoisted. For example, instead of performing the bank selection instruction for `LD A` of Fig. 1 inside the loop, we move the bank selection instruction outside of the loop when optimizing for speed (as depicted in Fig. 1(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 1. For a bank-sensitive basic block, there are at most two insertions, i.e., one before the fist 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

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

Table 1: Transformation for configuration (P_u, Q_u).

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 added to ensure that the bank register is set to Q_u upon exit (cf. again Table 1).

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 cost models.

A bank selection transformation $\mathbb{T}$ is optimal 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 the transparent basic blocks ($u \in T$).

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

Without loss of generality we devide the costs for a bank-sensitive basic block into the costs occuring 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. Other-

wise they return the cost c_u of a bank selection instruction.

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

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

For a transparent basic block the costs are defined by

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

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[1], i.e., $c_u \geq 0$. In our experiment, we have chosen a parameterizable cost metric:

$$c_u = \alpha \times \mathsf{SPEED_u} + \beta \times \mathsf{SPACE_u} \quad (10)$$

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 (i.e., a basic block introduced due to critical edge splitting, which must be transparent) or not, $\mathsf{SPEED_u}$ is defined as

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

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).

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

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

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:

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

A related problem was introduced in [9], 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 costs for local labeling and labeling of two related points into account. In [9] 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.

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

$\mathcal{T}$	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

Table 2: Cost matrices

4.3 Mapping to PBQP

PBQP [20, 4] is used to solve the discrete optimization problem of Eqn. (13). In the PBQP problem, the controlling variables P_u and Q_u of basic block u become Boolean vector variables $\vec{p}_u$ and $\vec{q}_u$. The length of $\vec{p}_u$ and $\vec{q}_u$ is $|\mathbb{D}|$, where each element of the vector represents an element in $\mathbb{D}$, i.e. Boolean vector $\vec{x} = (x_{b_0}, \ldots, x_{b_{m-1}}, x_{b_?})$ has zero and one variables where the first element corresponds to the first bank, the second element to the second, and so forth. The element before the last element corresponds to the last bank b_{m-1} and the last element denotes the unknown state of the bank register.

The costs of bank-sensitive basic blocks are modeled as cost-vectors and the costs of transparent basic blocks become cost matrices. As a result, the objective function of Eqn. (13) is transformed to the PBQP objective function:

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

Therein $\vec{n}_u$ and $\vec{e}_u$ are the cost functions $n\text{-}cost_u(P_u)$ and $e\text{-}cost_u(Q_u)$ in vector notation, and $(c_u \cdot \mathcal{T})$ is $t\text{-}cost_u(P_u, Q_u)$ in matrix notation.

The vector $\vec{n}_u$ is a zero vector. If the bank of the first bank-sensitive statement is $b_?$. Otherwise, it is of the structure

$\vec{p}_u$	b_0	b_1	...	b_{i-1}	b_i	b_{i+1}	...	b_{m-1}	$b_?$
$\vec{n}_u$	c_u	c_u	...	c_u	0	c_u	...	c_u	c_u

(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. Let's assume we optimize for speed. Basic block 4 in our motivating example (see Fig. 1(a)) is executed 90 times. The 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 90 cost units for the insertion before `LD A`. Thus $c_u = 90$. In vector notation, *n-cost*(4) is mapped to $\vec{n}_4 = \begin{pmatrix} 0 & 90 & 90 \end{pmatrix}$. 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 $b_?$, 90 cost units are imposed because a bank selection instruction needs to be inserted. Similarly, the cost vector $\vec{e}_u$ is constructed for bank sensitive basic block u, i.e.

$$\begin{array}{c|cccccccc} \vec{p}_u & b_0 & b_1 & \dots & b_{i-1} & b_i & b_{i+1} & \dots & b_{m-1} & b_? \\ \hline \vec{n}_u & c_u & c_u & \dots & c_u & 0 & c_u & \dots & c_u & 0 \end{array} \quad (16)$$

where b_i denotes bank lb_u. We have a bank insertion if the bank is not equal to Q_u or Q_u is set to $b_?$.

The structure of the cost matrix $\mathcal{T}$ for transparent basic blocks is given in Table 2(a) and is a direct mapping of Eqn. (9) in matrix notation with unit costs, i.e. $c_u = 1$. The rows of $\mathcal{T}$ correspond to a bank of variable P_u and the columns correspond to a bank of variable Q_u. If P_u and Q_u are not set to the same bank and Q_u is not $b_?$, we have costs of one, otherwise we have zero costs. We multiply matrix $\mathcal{T}$ with scalar c_u in Eq. (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,

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

where the constraint expressed in Eqn. (3) is mapped to matrix $\mathcal{R}$ shown in Table 2(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 Fig. 1 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

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

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.

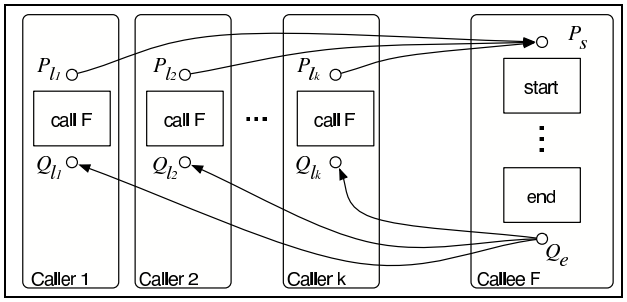

Figure 4: Interprocedural analysis.

The correctness constraints are

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

The objective functions $g = f + \Delta$ is to be solved to find the optimal 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.

If the inter-procedural optimization decides that a subroutine does not perform the bank selection instruction before the first bank sensitive statement of a function, then the caller of the function is forced to perform the bank selection instruction prior to the call. A similar mechanism is used to force the call site to set a bank after the call. This is beneficial for any kind of space optimization whereas the placement of bank selection instructions prior to a call are beneficial for speed optimizations.

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. For each call site l_i of subroutine F we add two correctness constraints as depicted in Fig. 4. 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

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

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

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

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

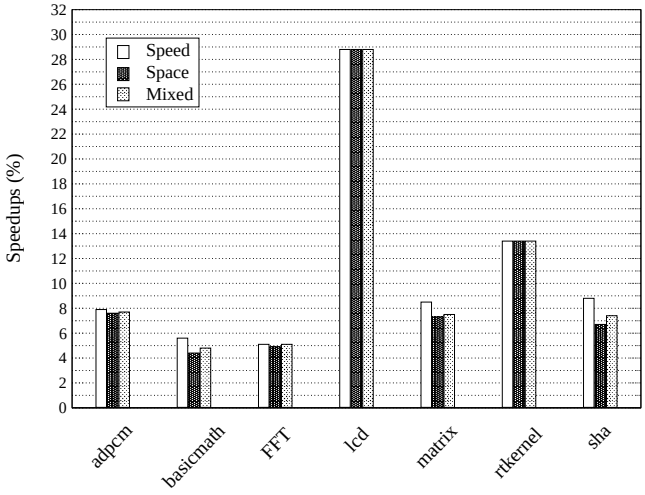

Figure 5: Speed-up.

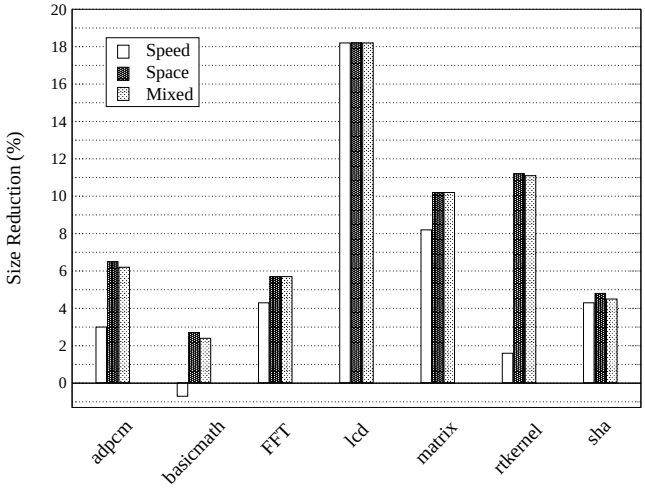

Figure 6: Space reduction.

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:

$$\Delta' = \Delta + \vec{p}_{x_i} \mathcal{R} \vec{p}_s + \vec{q}_e \mathcal{R} \vec{q}_{x_i}. \quad (20)$$

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

5. EXPERIMENTAL RESULTS

We evaluated our optimization for programs typically run on contemporary microcontrollers. Our sample included programs from the MiBench Embedded Benchmark Suite [7] and from DSPStone and we surveyed a microcontroller real-time kernel and common microcontroller driver routines. We conducted this experiment on a PIC16F877A microcontroller [13, 12]. We applied different optimization objectives to show the versatility of our approach.

The PIC family of midrange microcontrollers (cf. [12]) constitutes a RISC-based Harvard architecture with instruction sizes of 12, 14 or 16 bits, and a data-bus that is 8 bit wide. The PIC16F877A microcontroller provides 8kB of program memory and 368B of data memory spread over four banks [13]. The MiBench programs that we could fit onto this microcontroller included "basicmath", "sha", 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 from [14].

Our experimental setup is depicted in Fig. 7. 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 [22] 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. It is important to note that the extended assembly file contained the bank selection instructions as they were 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 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 goal of the optimizer was then 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. 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 it 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 [18]. 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 3, we determined the overall static instruction count ("Total") together with the number of bank selection instructions ("BSL") for each benchmark. We de-

				Instruction count		Cycle count		Space reduction		Improvement	
Benchmark	Objective	α(%)	β(%)	Total	BSL	Total	BSL	Total%	BSL%	Total%	BSL%
adpcm	HiT	n/a	n/a	6081	795	6.0e+05	76927	n/a	n/a	n/a	n/a
	Speed	100	0	5898	558	5.6e+05	33070	3.0	29.8	7.9	132.6
	Space	0	100	5683	397	5.6e+05	34322	6.5	50.1	7.6	124.1
	Mixed	50	50	5707	411	5.6e+05	33966	6.2	48.3	7.7	126.5
basicmath	HiT	n/a	n/a	1844	248	5856	1023	n/a	n/a	n/a	n/a
	Speed	100	0	1857	237	5545	712	-0.7	4.4	5.6	43.7
	Space	0	100	1794	198	5608	775	2.7	20.2	4.4	32.0
	Mixed	50	50	1799	199	5590	757	2.4	19.8	4.8	35.1
FFT	HiT	n/a	n/a	2661	336	53254	6491	n/a	n/a	n/a	n/a
	Speed	100	0	2546	206	50683	3920	4.3	38.7	5.1	65.6
	Space	0	100	2508	183	50775	4012	5.7	45.5	4.9	61.8
	Mixed	50	50	2510	185	50683	3920	5.7	44.9	5.1	65.6
lcd	HiT	n/a	n/a	307	65	72707	16285	n/a	n/a	n/a	n/a
	Speed	100	0	251	9	56431	9	18.2	86.2	28.8	1.8e+05
	Space	0	100	251	9	56431	9	18.2	86.2	28.8	1.8e+05
	Mixed	50	50	251	9	56431	9	18.2	86.2	28.8	1.8e+05
matrix	HiT	n/a	n/a	401	62	716	105	n/a	n/a	n/a	n/a
	Speed	100	0	368	27	660	49	8.2	56.5	8.5	114.3
	Space	0	100	360	21	667	56	10.2	66.1	7.3	87.5
	Mixed	50	50	360	21	666	55	10.2	66.1	7.5	90.9
rtkernel	HiT	n/a	n/a	2097	422	2.6e+08	4.9e+07	n/a	n/a	n/a	n/a
	Speed	100	0	2063	343	2.3e+08	1.8e+07	1.6	18.7	13.4	169.9
	Space	0	100	1862	187	2.3e+08	1.8e+07	11.2	55.7	13.4	169.8
	Mixed	50	50	1864	189	2.3e+08	1.8e+07	11.1	55.2	13.4	169.9
sha	HiT	n/a	n/a	3017	170	8.2e+05	153749	n/a	n/a	n/a	n/a
	Speed	100	0	2886	35	7.5e+05	87390	4.3	79.4	8.8	75.9
	Space	0	100	2873	26	7.7e+05	102552	4.8	84.7	6.7	49.9
	Mixed	50	50	2880	29	7.6e+05	97498	4.5	82.9	7.4	57.7

Table 3: Experimental results for the PIC microcontroller benchmark programs.

termined the cycle counts induced by these instruction-categories 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 parameter values for α and β are depicted in Table 3. In terms of Eqns. (11) and (12), we set bsl-cycles = bsl-size = jump-cycles = jump-size = 1. The rightmost columns in Table 3 depict the memory footprint reduction and the resulting performance improvement achieved by our optimization. The figures clearly reflect the goals of the different optimization objectives: optimizing for space results in the lowest number of issued bank selection instructions, whereas optimization for speed minimizes instruction cycles. Optimization 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 3 that the amount of reduction of the program memory footprint (corresponding to the overall instruction count) is between 2.7% and 18.2% when we optimize for *space*. The reduction of bank selection instructions is then between 20.2% and 86.2%. If we optimize for *speed*, the achieved overall improvement is between 5.1% and 28.8%, and the improvement with respect to the execution of bank selection instructions alone is between 65.6% and 180000%. None of the benchmark programs required the PBQP solver to apply an RN reduction; this means that our optimization always achieved the optimal solution. The overall speedup is shown in the bar chart of Fig. 5, and the program-size reduction of our bank selection optimization is shown in Fig. 6.

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 have formulated the bank selection placement problem as a discrete optimization problem. Optimization objectives are modeled as cost metrics and allow parameterizable trade-offs between several objectives, such as speed and space. We have provided empirical evidence that our method performs well for embedded systems architectures. We have demonstrated an efficient algorithm using Partitioned Boolean Quadratic Programming for optimal bank selection placement.

We have conducted experiments with programs from the DSPStone and MiBench benchmark suite to demonstrate the practicality of our approach. To show the feasibility of our approach, we have implemented a toolchain for the Microchip PIC 16F877a microcontroller. This tool-chain op-

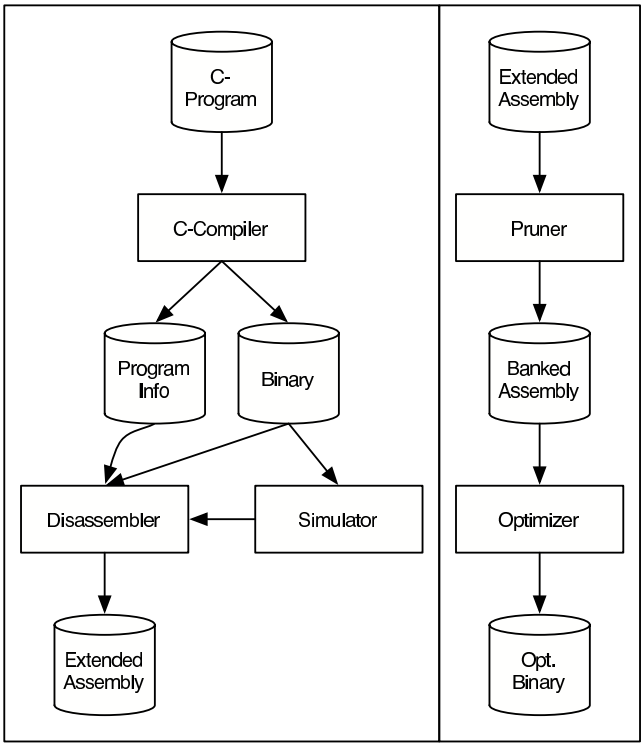

Figure 7: Toolchain.

timizes the bank selections of binaries. For the surveyed programs we achieved speedups to 28.8% and code size reductions to 18.2% where the base-line is a state-of-the-art C-compiler for the PIC 16F877a.

Acknowledgements

We would like to thank Merrilee Robb and Wei-Ying Ho for proofreading the manuscript. We would like to thank Sanjay Chawla for presenting paper [9] in our algorithmic reading group.

7. REFERENCES

[1] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign, CODES, Estes Park (Colorado)*, May 2002.

[2] Jeonghun Cho, Yunheung Paek, and David Whalley. Fast Memory Bank Assignment for Fixed-Point Digital Signal Processors. *ACM Transactions on Design Automation of Electronic Systems*, 9(1):52–74, 2004.

[3] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 138–147, New York, NY, USA, 2000. ACM Press.

[4] Erik Eckstein. *Code Optimizations for Digital Signal Processors*. PhD thesis, Institute of Computer Languages, Compilers and Languages Group, Vienna University of Technology, 2003.

[5] Gartner Dataquest. 2003 Microcontroller Market Share and Unit Shipments, July 2004.

[6] Gartner Dataquest. Top Companies Revenue from Shipments of 8-bit MCU — All Applications, April 2005.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[8] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen-Mei Hwu. Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 247–256, New York, NY, USA, 1993. ACM Press.

[9] Jon M. Kleinberg and Eva Tardos. Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 14–23, 1999.

[10] R. Leupers and D. Kotte. Variable Partitioning for Dual Memory Bank DSPs. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 1121–1124, 2001.

[11] Lian Li, Lin Gao, and Jingling Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *PACT '05: Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, 2005.

[12] Microchip Technology Inc. PICmicro Mid-Range MCU Family Reference Manual, 1997.

[13] Microchip Technology Inc. PIC16F87XA Data Sheet, 2003.

[14] MicrochipC.com PIC Micros and C. `http://www.microchipc.com/`, 2006.

[15] Erik Nystrom and Alexandre E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 103–114, 1998.

[16] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, 2001.

[17] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. 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, 2000.

[18] PICC ANSI C Compiler. `http://www.htsoft.com/`, 2006.

[19] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In *ASPLOS-VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, New York, NY, USA, 1996. ACM Press.

[20] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *LCTES-SCOPES '02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 139–148. ACM, 2002.

[21] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 388–392, 1995.

[22] The Gpsim SW Simulator for PIC Microcontrollers. `http://www.dattalo.com/gnupic/gpsim.html`, 2006.

[23] Sumesh Udayakumaran and Rajeev Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 276–286. ACM Press, 2003.

[24] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1264–1269, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Xiaotong Zhuang, Santosh Pande, and John S. Greenland Jr. A Framework for Parallelizing Load/Stores on Embedded Processors. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–79. IEEE Computer Society, 2002.

[26] Qingfeng Zhuge, Bin Xiao, and Edwin Hsing-Mean Sha. Variable Partitioning and Scheduling of Multiple Memory Architectures for DSP. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 332, Washington, DC, USA, 2002. IEEE Computer Society.