Efficient Use of Invisible Registers in Thumb Code *

Arvind Krishnaswamy Rajiv Gupta
The University of Arizona
Department of Computer Science
Tucson, Arizona 85721

Abstract

The ARM processor is a dual width ISA processor that provides a 16-bit Thumb instruction set in addition to the 32-bit ARM instruction set. The compromises made in designing the Thumb instruction set leads to significantly increased instruction counts. This increase is in part due to the fact that only half of the register file is visible to most instructions in Thumb code. In this paper we address this inefficiency by providing a new instruction, the `SetMask` *instruction, using which the compiler can change the visible subset of registers at any program point. Thus, through the use of this instruction the compiler can make use of all registers in all instructions. We present compiler techniques for allocating invisible registers and introducing* `SetMask` *instructions in a manner that the number of introduced instructions is minimized so that the increase in code size is insignificant. We implement this new instruction using the Dynamic Instruction Coalescing Framework which enables the* `SetMask` *instruction to have zero execution time cost. Our techniques eliminated 11.7% of MOV instructions from Thumb code while causing negligible code size increase.*

1 Introduction

More than 98% of all microprocessors are used in embedded products, the most popular 32-bit processors among them being the ARM family of embedded processors [22]. The ARM processor core is used both as a macrocell in building application specific system chips and standard processor chips [21, 18, 3]. In the embedded domain, in addition to having good performance, applications must execute under constraints of limited memory. ARM supports dual width ISAs that are simple to implement and provide a tradeoff between code size and performance. In prior work [7] we studied the characteristics of ARM and Thumb code and showed that for some embedded applications the Thumb code size was 29.8% to 32.5% smaller than the corresponding ARM code size. However, it was also observed that there was an increase in instruction counts for Thumb code which was typically around 30% [7]. We studied the instruction sets and then compared the Thumb and ARM code versions to identify the causes of performance loss. The reasons we identified fall into two categories: *Global inefficiency* - Global inefficiency arises due to the fact that only half of the register file is *visible* to most instructions in Thumb code. *Peephole inefficiency* - Peephole inefficiency arises because pairs of Thumb instructions are required to perform the same task that can be performed by individual ARM instructions.

The peephole inefficiency problem has been addressed by developing a *dynamic instruction coalescing* framework that consists of instruction set, microarchitecture, and compiler support that enables pairs of instructions in Thumb code to be replaced by single ARM instructions [6] . The compiler enables coalescing by generating AXThumb code which is Thumb instructions extended with *Augmenting eXtensions* (AX). The coalescing is performed in a manner that no bubbles in the pipeline are introduced. One could generate a mixed binary using both ARM and Thumb instructions; however, the overhead of explicit switching between 16-bit mode and 32-bit mode for short sequences negates the benefit of mixed code [6].

In this paper we address the *global inefficiency* problem by providing architectural support for exposing the *invisible* registers to the compiler in Thumb mode. The registers in the register file are viewed as pairs of corresponding registers in lower and upper half of the register file. At any program point, one register from each pair is visible to the compiler. However, by executing a special `SetMask` instruction the set of registers that are visible to the compiler can be changed. Thus, the compiler can use all registers as long as it makes appropriate use of `SetMask` instructions. The key challenges for the compiler are to allocate and assign registers and then generate minimal number of `SetMask` instructions. The reason for careful generation of `SetMask` instructions is the increase in code size that

*Supported by grants from Intel, Microsoft, IBM, and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, and EIA-0080123 to the Univ. of Arizona.

it may cause thus negating the benefit of Thumb code over ARM code. Another significant feature of our solution is that it makes use of the same microarchitectural extensions that were used to address peephole inefficiencies. Thus not only can both peephole and global inefficiencies be handled together by this architecture, but the cost of executing the augmenting instructions such as `SetMask` is zero. Thus, the impact of using the `SetMask` instructions is only a decrease in instruction count by avoiding spill code. By effective use of `SetMask` instructions we can maintain compact code size and improve execution time performance.

2 Dynamic Instruction Coalescing

2.1 Peephole Optimization

Our prior work [6] introduced a dynamic instruction coalescing framework which enabled removal of peephole inefficiencies from Thumb code. The impact of coalescing was to enable translation of a pair of Thumb instructions into a single ARM equivalent at runtime. To achieve this goal the Thumb instruction set was enhanced by incorporating *Augmenting eXtensions* (AX). Augmenting instructions are a new class of instructions which are entirely handled in the decode stage of the processor and do not go through the remaining stages of the pipeline. Each AX instruction is coalesced with the following non-AX instruction in the program, in the decode stage of the processor where the translation of Thumb instructions into ARM instructions takes place. The *compiler* replaces patterns of Thumb instructions by equivalent sequences of AXThumb instructions. The *decode stage* is redesigned to detect augmenting instructions and perform coalescing to generate more efficient ARM instructions for execution. When coalescing is performed, no additional pipeline bubbles are introduced as instruction fetching does not fall behind. When two instructions are coalesced during execution of AXThumb code, two additional Thumb instructions are available for decoding in the very next cycle. By placing the responsibility of identifying instruction coalescing opportunities on the compiler, AX enables us to achieve coalescing using simple modifications to the decode stage. While a compiler can easily recognize coalescing opportunities, and appropriately mark them using AX instructions, the hardware cannot do so either easily or safely.

```
ARM:      sub reg1, reg2, lsl #2
Thumb:    lsl rtmp, reg2, #2
          sub reg1, rtmp
AXThumb:  setshift lsl #2
          sub reg1, reg2
```

To illustrate the key concepts of the above approach lets consider an example. In the code above we show an ARM instruction which shifts the value in `reg2` before subtracting it from `reg1`. Since the shift cannot be specified as part of another Thumb ALU instruction, two Thumb instructions are required to achieve the effect of one ARM instruction. We would like to coalesce the two 16-bit instructions into one 32-bit instruction. While coalescing is relatively easy to carry out, detecting a legal opportunity for coalescing by examining the two Thumb instructions is in general impossible to carry out at run-time with simple hardware. In our example, the Thumb code uses a temporary register `rtmp`. If instruction coalescing is performed, `rtmp` is no longer needed; therefore its contents will not be changed. Hence, at the time of coalescing, the hardware must also determine that the contents of register `rtmp` will not be used after the Thumb sequence. Clearly this is in general impossible to determine since the next read or write reference to register `rtmp` can be arbitrarily far away. Since the coalescing opportunity cannot be detected in hardware we rely on the compiler to recognize such opportunities and communicate them to the hardware through the use of *Augmenting eXtensions* (AX). In the AXThumb code shown above, the first instruction is an augmenting instruction which is not executed; it is always coalesced in the decode stage with the instruction that immediately follows it, to generate a single ARM instruction for execution. In the above example, the augmenting instruction `setshift` merely carries the shift type and shift amount, which is incorporated in the subsequent instruction to create the required ARM instruction.

It should be noted that the code size of all three instruction sequences is the same (i.e., 32 bits). However, only the AXThumb sequence satisfies the desired criteria as it results in the execution of a single equivalent ARM instruction and is made up of 16-bit instructions. Thus, the AXThumb code is 16-bit code that runs like the ARM code. The execution of Thumb code and AXThumb code are shown in Figure 1. The AX instruction is processed in parallel with the preceding Thumb instruction and thus it does not introduce an extra cycle of execution time. The detailed design of the dynamic coalescing microarchitecture can be found in [6].

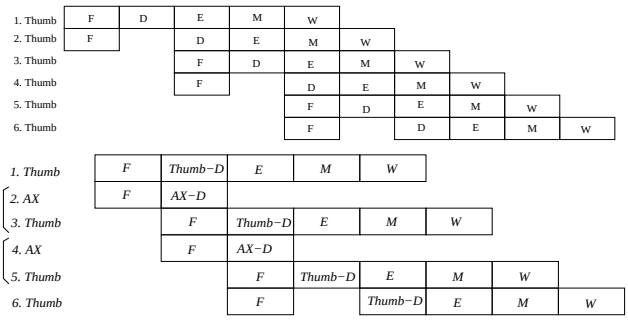

Figure 1. Thumb vs. AXThumb.

2.2 Global Optimization

In Thumb instructions the register specifier field is typically 3 bits while it is 4 bits in all ARM instructions. Thus, in Thumb mode the lower half of the register file (i.e., $R0 \cdots R7$) can be freely accessed while the upper half (i.e., $R8 \cdots R15$) of the register file is accessible only by very few instructions (e.g., MOVs).

The dynamic coalescing framework can be used to enable the higher order registers visible to all Thumb instructions. To achieve this goal we take the following approach. We view the register file containing 16 registers as consisting of 8 register pairs – $(r0, r8)$, $(r1, r9)$, $\cdots$ $(r7, r15)$. Only 8 registers are visible at a time such that exactly one register is visible from each pair at any point in time. For each register pair, the register from the pair that is visible at any point in time needs to be set. For this purpose we provide the `SetMask` instruction. This instruction has an 8 bit operand where each bit in the operand specifies the leading bit of the register specifier. If the leading bit for a pair is 0 then the lower order register is visible while if the leading bit is 1 then the higher order register from the pair is visible. Each time in the program when the set of visible registers needs to be changed, a single `SetMask` instruction is executed to achieve this goal.

The execution of the `SetMask` instruction is achieved without adding additional execution cycles using the dynamic coalescing framework. A `SetMask` instruction is processed in the decode stage in parallel with the preceding Thumb instruction in the same manner as other AX instructions are accessed as described in the preceding section. The decode stage saves the bits specified in the `SetMask` and uses these bits to interpret the register specifier bits in future Thumb instructions till the next `SetMask` instruction is encountered. While the above approach greatly reduces the constraints on use of registers by Thumb instructions, one minor constraint still remains – a Thumb instruction cannot simultaneously reference both registers from a register pair. The register allocator must take this constraint into account during register assignment.

Figure 2 shows how register operands use the bitmask set by the `SetMask` instruction. The register file needs to be modified to accommodate register operand access without delays. The register file is organized such that each row contains the high and low registers. The lower 3 bits of the register specifier are used to index the register file as well as the bitmask. In 16-bit state the bit selected from the bitmask is used to select the low or high register contents of the selected row. In 32-bit ARM state the MSB of the register specifier selects the low or high register. Both the bitmask and register file are indexed in parallel to avoid introducing delays during register file access.

Let us consider a simple example of Figure 3 that illustrates the advantage of using `SetMask` instruction. In this

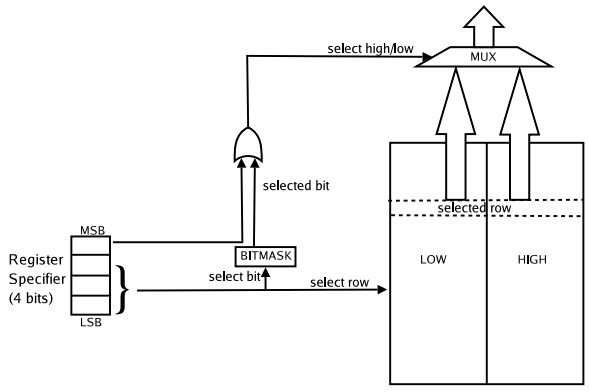

Figure 2. Register Operand Access

[a in R0; c in R5;] [t - temporary]			

t = c + 5
a = a + t
	MOV R10, R0
	ADD R0, R5, #5
	ST R5, $< c >$
	MOV R5, R10
	ADD R0, R0, R5
	LD R5, $< c >$
		ADD R10, R5, #5
		ADD R0, R0, R10
			SetMask 0x04
			ADD R2, R5, #5
			ADD R0, R0, R2

(a) IR	(b) Thumb	(c) ARM	(d) AX Thumb

Figure 3. Use of `SetMask`.

example, we assume that registers $R0$ and $R5$ are available to hold the values of variables a and c respectively immediately preceding and following the code fragment. The variable t is a temporary which is computed and consumed within the code fragment. Let us assume that other than $R0$ and $R5$ the only register available is $R10$ while all other lower and higher order registers are already occupied by other variables. Under these assumptions we show the generated code sequences for ARM, Thumb, and AXThumb. As we can see, in case of ARM only two instructions are generated where $R10$ is used for the temporary t. When `SetMask` instruction is used, AXThumb code generated is similar to the ARM code except that the `SetMask` is used so that make $R10$ visible. The number of cycles it takes to execute the ARM and AXThumb codes is the same as the `SetMask` instruction does not cost extra cycles. Now we finally look at Thumb code in which case $R10$ is used to spill the value in $R0$ since $R10$ cannot be directly accessed by the ADD. As we can see, spill code must be generated that causes extra load and store instructions to be generated – we have observed these effects in the Thumb code generated by the `gcc` compiler.

3 Exploiting Exposed Registers

It is clear that by using `SetMask` instructions, the execution time of code can be improved in comparison to Thumb code. However, there is another important issue that we must deal with. We would like to ensure that the code size of AXThumb code is also small. While extra `SetMask` instructions are needed, by using higher order

registers some of the spill code is eliminated. A *naive approach* for introducing `SetMask` instructions may be to add a `SetMask` instruction before and after each reference to a higher order register. However, this approach adds too many instructions causing a significant increase in the code size as shown in Figure 4. Therefore in this section we develop compiler algorithms for carefully introducing `SetMask` instructions.

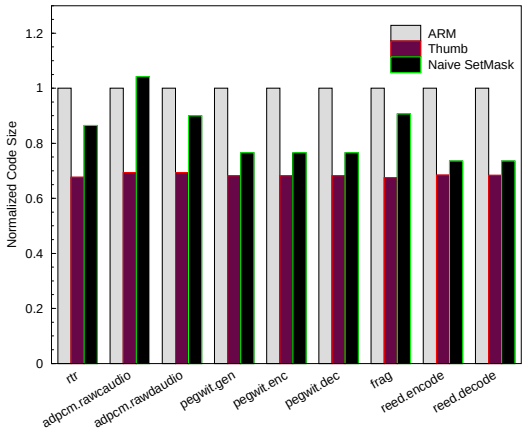

Figure 4. Normalized Code Size

In the rest of the paper we will use the following notation. $(R, \bar{R})$ refers to a register pair. While a `SetMask` instruction selects a register to make visible from each pair, when we use the notation `SetMask` R we will be referring to a `SetMask` instruction which makes R visible while keeping the visible registers from remaining pairs the same.

We assume that the Thumb register allocation is performed so that it uses all registers. After register allocation has been performed, the resulting code is examined and `SetMask` instructions are appropriately introduced. The algorithm for `SetMask` placement consists of three main steps. The first step separately determines *initial placement points* for `SetMask` R and `SetMask` $\bar{R}$ instructions corresponding to each register pair $(R, \bar{R})$. The second step finds the *placement ranges* for each `SetMask` R and `SetMask` $\bar{R}$ instruction which represent all points where an instruction can be placed. The third and final step *coalesces* multiple `SetMask` instructions referring to different register pairs into a single `SetMask` instruction that simultaneously changes the visibility of multiple registers and determines the `final placement` of `SetMask` instructions. The placement ranges identified in the second step are used to identify final placement points which enable maximal coalescing so that fewer `SetMask` instructions are introduced. Next we discuss these steps in detail.

Initial Placement Points

Given a register pair $(R, \bar{R})$, a simple way of introducing a `SetMask` instructions for this register pair is as follows. Immediately preceding an instruction that refers to R we can introduce a `SetMask` R instruction which makes R visible while immediately preceding an instruction that refers to $\bar{R}$ we introduce a `SetMask` $\bar{R}$ instruction which makes $\bar{R}$ visible. Clearly this approach will introduce a lot of `SetMask` instructions – preceding each instruction as many `SetMask` instructions will be introduced as the number of registers referenced by the instruction. The goal of our algorithm (all three steps) is to reduce this number.

In this first step we separately consider each register pair $(R, \bar{R})$ and simply try to eliminate unnecessary `SetMask` R and `SetMask` $\bar{R}$ instructions that are introduced by the above simple strategy. The basic idea of this step is illustrated in Figure 5. The first figure shows all references to R and $\bar{R}$ which are marked by ovals. The second figure shows the initial placement points of `SetMask` R and `SetMask` $\bar{R}$ instructions which are marked by squares. As we can see, there is no `SetMask` $\bar{R}$ introduced preceding the second $\bar{R}$ reference. This is because once we introduce `SetMask` $\bar{R}$ before the first $\bar{R}$ reference, the one before the second $\bar{R}$ becomes redundant.

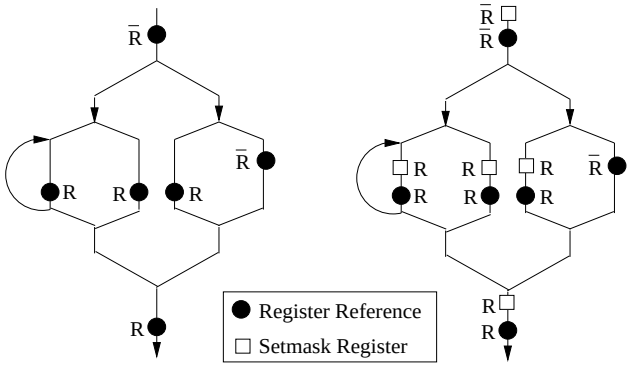

Figure 5. Initial Placement Points.

The determination of initial placement points for a given $(R, \bar{R})$ pair is made using the analysis shown in Figure 6. Forward *must availability* analysis is used to determine whether or not there is a need to introduce a `SetMask` $R/\bar{R}$ instruction preceding a $R/\bar{R}$ reference. If `SetMask` $R/\bar{R}$ instruction has been executed along all paths prior to reach a reference $R/\bar{R}$ such that $R/\bar{R}$ is already visible, there is no need to introduce a `SetMask` $R/\bar{R}$ instruction before this reference. The $(R, \bar{R})Avail$ sets are computed for all program points as shown in Figure 6 and then using this information the initial placement points are determined in form of $(R, \bar{R})Initial$ sets.

Placement Ranges

The previous step determined the latest points at which `SetMask` $R/\bar{R}$ instructions can be placed as they are placed just immediately before $R/\bar{R}$ references. However, we do not simply place them at the initial placement points.

Forward Availability

Initialize:

$(R, \bar{R})Avail_n(s) := (R, \bar{R})Avail_x(s) := \phi$

Solve:

$$(R, \bar{R})Avail_n(s) := \bigcap_{p \in Pred(s)} (R, \bar{R})Avail_x(p)$$

$$(R, \bar{R})Avail_x(s) := \begin{cases} (R, \bar{R})Avail_n(s) & R \notin Ref(s) \ \wedge \ \bar{R} \notin Ref(s) \\ \{R\} & R \in Ref(s) \\ \{\bar{R}\} & \bar{R} \in Ref(s) \end{cases}$$

Initial Placement Points

$$(R, \bar{R})Initial_n(s) := \begin{cases} \{SetMask\ R\} & R \in Ref(s) \ \wedge \ R \notin (R, \bar{R})Avail_n(s) \\ \{SetMask\ \bar{R}\} & \bar{R} \in Ref(s) \ \wedge \ \bar{R} \notin (R, \bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

$$(R, \bar{R})Initial_x(s) := \phi$$

Figure 6. Step 1: Initial Placement Points Determination.

This is because we would like to combine instructions corresponding to different register pairs and place them as part of a single `SetMask` instruction – after all, the `SetMask` instruction being supported allows us to simultaneously affect the visibility of registers within each register pair. We would like to identify program points where multiple initial instructions can be coalesced and placed. Thus, starting from the initial point placements, we perform analysis that determines all the places where the instructions can be placed, i.e. we expand initial placement points into *placement ranges*.

The placement range *corresponding to a* `SetMask` $R/\bar{R}$ *instruction is a contiguous portion of the control flow graph such that the instruction can be placed at any point in the range. Each range has distinguishing* earliest points *and* latest points. *An earliest (latest) point belonging to a placement range is a point such that none of its predecessor (successor) points belong to the placement range*.

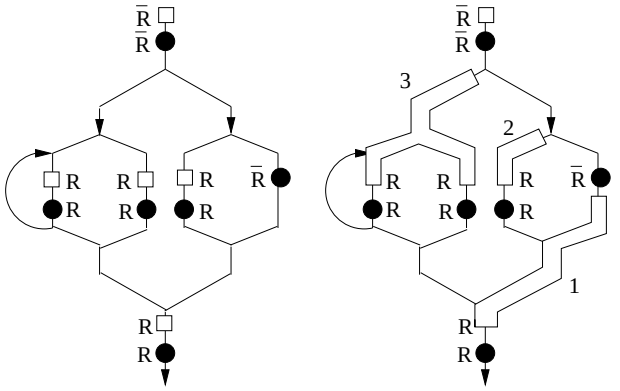

Figure 7. Placement Ranges.

Before we present the detailed analysis for identifying placement ranges, we illustrate this process by continuing with the example we used to illustrate initial placement point identification. In Figure 7, placement ranges corresponding to the initial placement points are shown. Lets look at the three ranges marked 1, 2, and 3 in detail as they demonstrate the various cases that we must take into account in designing the analysis:

- The range marked 1 indicates that while the latest point for placement of `SetMask` R instruction was immediately prior to the reference to R, its earliest placement point is the point that immediately follows the reference to $\bar{R}$. In fact, this instruction can be safely placed at any point along the range that extends from the earliest point to the latest point. There is no need to place this instruction along the paths that merge into this range because `SetMask` R is already available along those paths.
- The range marked 2 extends to the point just below the split point. The earliest point cannot extend above the split point because if `SetMask` R is placed above the split point, the availability of `SetMask` $\bar{R}$ preceding the first $\bar{R}$ at the second $\bar{R}$ will be disrupted.
- The range marked 3 is interesting in that it includes two distinct initial (latest) placement points of `SetMask` R. However, it results in a single earliest point. In other words, if we place the instruction at the earliest point we need one instruction but if we place it at the latest points we need two instructions. Note that in ranges marked 1 and 2 there was a single latest point and single earliest point.

Placement Ranges

Initialize:

$$(R,\bar{R})Range_{n/x}(s) := (R,\bar{R})Initial_{n/x}(s)$$

Solve:

$$(R,\bar{R})Range_x(s) := \bigcap_{q \in Succ(s)} (R,\bar{R})Range_n(q)$$

$$(R,\bar{R})Range_n(s) := (R,\bar{R})Range_x(s) - (R,\bar{R})Kill(s)$$

where,

$$(R,\bar{R})Kill(s) := Kill(s,R) \cup (Kill(s,\bar{R})$$

$$Kill(s,R) := \begin{cases} \{SetMask\ R\} & \bar{R} \in Ref(s) \ \vee \ R \in (R,\bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

$$Kill(s,\bar{R}) := \begin{cases} \{SetMask\ \bar{R}\} & R \in Ref(s) \ \vee \ \bar{R} \in (R,\bar{R})Avail_n(s) \\ \phi & otherwise \end{cases}$$

Figure 8. Propagating Initial Placement Points Backwards to Build Placement Ranges.

Based upon the illustration above, we are now ready to state the conditions under which a placement point of `SetMask` R can continue to expand into a live range through backward propagation. We can continue to extend the range of `SetMask` R backwards along program points as long as a reference to $\bar{R}$ is not encountered and a point is not reached where `SetMask` R is already available according to the analysis carried out in Step 1 of our algorithm. Based upon these conditions we define the $Kill$ sets used to stop propagation. When a split point is reached, we use the intersection operator to decide whether to continue propagation. As we can see, the intersection operation will correctly prevent and allow propagation above split points for formations of ranges 2 and 3 respectively in our example. The detailed analysis equations are given in Figure 8. The results of this analysis are interpreted as follows – for each program point that belongs to a range for `SetMask` $R/\bar{R}$, the set $(R,\bar{R})Range$ is set to {`SetMask` $R/\bar{R}$}; otherwise it is set to empty. As we can see, the results of analysis of Step 1 play a crucial role in Step 2. First, the initialization of $(R,\bar{R})Range$ values at program points is based upon the initial points identified in Step 1. Second, the $Kill$ sets needed during propagation require the use of $(R,\bar{R})Avail$ information also computed during Step 1. The backward propagation conditions identified above are used by the $Kill$ sets and intersection operator is used at split points.

Coalescing and Final Placement

Now we know the placement ranges of all `SetMask` instructions for each register pair. The goal of this step is to choose final placement points of `SetMask` instructions in a way that enables coalescing of `SetMask` instructions belonging to different register pairs. Such coalescing will reduce the number of instructions introduced. The continuation of our example illustrates the choices. In Figure 9, the figure on the left considers the situation in which `SetMask` instructions are only needed for $(R,\bar{R})$. Therefore no coalescing opportunities exist and the choice as shown is made. Now let us assume that there are points at which `SetMask` instructions for $R1$, $R2$ and $R3$ are definitely needed at the points shown in the figure on the right. The `SetMask` instructions for R can be coalesced with them resulting in the placement shown.

In order to develop an algorithm for this final step we make the following observation. Given a placement range for say R, depending upon the selection of placements points the number of `SetMask` R instructions needed to handle this range can vary (e.g., for range marked 3 of Figure 7, we may need one or two instructions depending upon the placement points). Moreover, when all register pairs are considered the best overall choice for R need not be the one which requires minimum number of `SetMask` R instructions. This is because the cost of placing `SetMask` R instructions depends upon whether or not they can be coalesced with similar instructions for other registers.

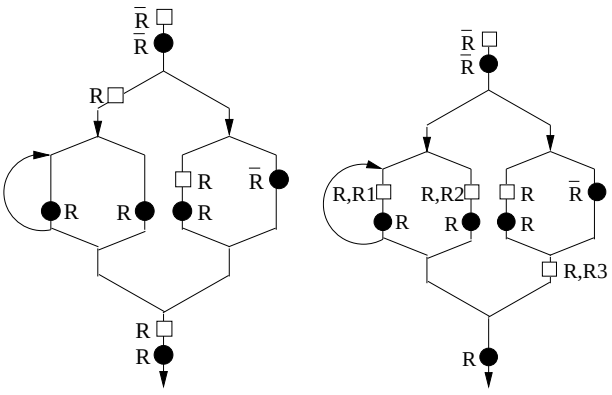

Figure 9. Final Placement Points.

To explore coalescing opportunities, we develop a for-

mulation of placement point selection where the placement point decisions for all register pairs can be made simultaneously. The key first step in this formulation is the decomposition of placement ranges into *placement paths*.

We define a placement path *as a contiguous section of the placement range extending from a single earliest point to a single latest point.*

Following the decomposition we construct an *overlap graph* which captures all the overlapping relationships amongst the placement paths of all register pairs – the nodes in the graph represent individual placement paths and edges connect pairs of nodes that overlap with each other. This graph is then used to guide the placement decisions. A `clique` in the graph represents a group of placement paths that can be all covered by a single `SetMask` instruction. We iteratively select cliques from the graph and introduce `SetMask` instructions till all placement paths have been covered. The number of instructions introduced equals the number of cliques selected to cover the entire graph.

Next we illustrate the above steps of the algorithm using an example. In Figure 10, a *placement range* for `SetMask` R is shown. This placement range has three earliest points and one latest point. Upon decomposition, this placement range gives rise to three placement paths (P_0, P_1 and P_2). Now let us see how the *overlap graph* is constructed. Three nodes corresponding to the three paths are created and connected to each other as the three paths overlap. Let us consider presence of additional placement paths corresponding to registers $R1$, $R2$, and $R3$ giving rise to the overlap graphs shown in Figure 11. Now let see how the `SetMask` instructions are introduced. Cliques are used to cover the overlap graph and each clique corresponds to a `SetMask` instruction whose form is determined by the placement paths contained in the clique. This step is illustrated in Figure 12. The cliques chosen and the corresponding instructions introduced are shown. Note that in each case we have chosen the minimum number of cliques needed to cover the overlap graph. Minimum number of cliques correspond to minimum number of `SetMask` instructions.

Next we discuss how the cliques are selected. A greedy algorithm may be used that selects the *maximal clique* at each step. However, if there are multiple cliques of the same size that share nodes, then we need to decide which clique to pick as the choice will effect the total number of instructions introduced. For example, in the first overlap graph of Figure 13, there are two maximal cliques with three nodes – (P_1, P_2, R_2) and (P_0, P_1, P_2). If we select the first clique the graph is covered by two cliques while if we choose the second clique we need three cliques to cover the graph. We can further refine the greedy heuristic to choose between multiple maximal cliques. We can determine the minimum degree across nodes neighboring a maximal clique after the clique has been removed. The higher the degree the better

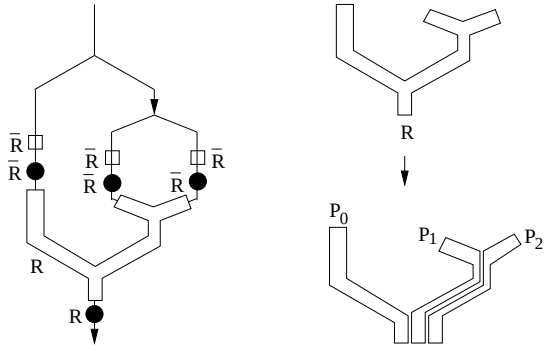

Figure 10. Splitting Placement Ranges into Placement Paths.

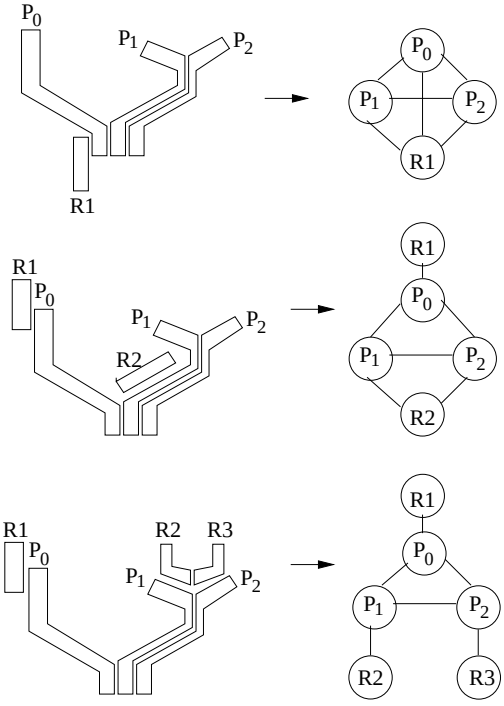

Figure 11. Overlap Graph.

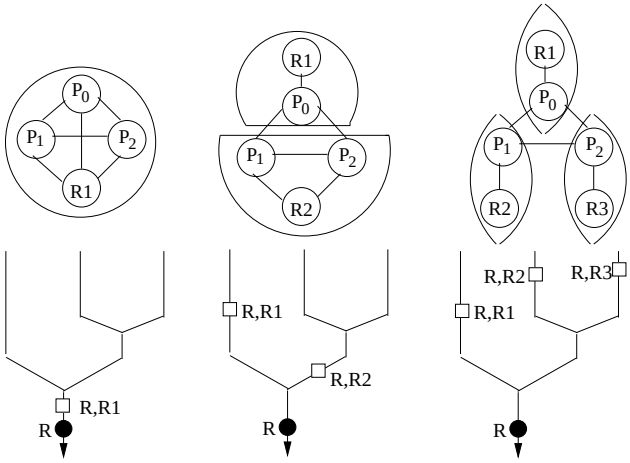

Figure 12. Clique Selection and Final Placement.

it is because higher degree is likely to translate into larger future cliques being available. This approach will have the desired result of selecting (P_1, P_2, R_2) in the first example shown below. While we use the above approach, it is important to point out that this is a heuristic and thus it will not guarantee optimal results. Even if there is a unique maximal clique, always picking maximal clique does not necessarily result in fewest instructions. In the second example shown in Figure 13, if the do not choose the maximal clique (P_0, P_1, P_2) we can cover the graph using three cliques while if we choose the maximal clique (P_0, P_1, P_2) we need four cliques to cover the graph.

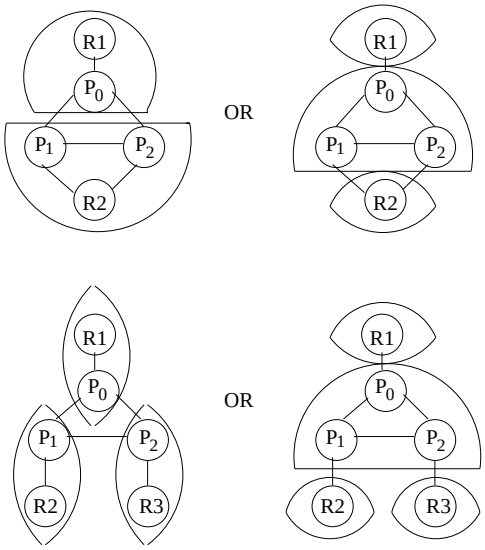

Figure 13. Clique Selection.

We would also like to mention that finding the maximal cliques is quite straightforward. We can examine each program point and count the number of placement paths to which that point belongs. This approach will identify all cliques and thus we can identify all maximal sized cliques. The cliques can then be prioritized by going back to the overlap graph and accessing the impact of selecting each of these cliques on degrees of neighboring nodes.

Finally we summarize the details of Step 3 in Figure 14. As we can see, first placement ranges are decomposed into placement paths. Next the overlap graph is constructed and then one by one cliques and selected and removed from the overlap graph and corresponding `SetMask` instructions are inserted in the program.

4 Experimental Evaluation

The goal of our experiments is two fold. First we would like to determine the benefit in performance that results from making use of `SetMask` instructions. Second we would like to determine the effectiveness of our presented algorithms in limiting the increase in code size. To carry

Final Placement Points

Decompose all *Placement Ranges* into *Placement Paths*.
Construct *Overlap Graph*:
  Nodes correspond to placement paths; and an Edge between a pair of nodes indicates that the corresponding paths overlap.
While *Overlap Graph* is not empty **do**
  Find *Maximal Cliques*.
  Select highest priority *Maximal Clique*.
  Remove selected clique from the *Overlap Graph*
    and insert *coalesced* `SetMask` instruction.
endwhile

Figure 14. Coalescing and Final Placement.

out this experimentation we implemented the described techniques in our simulation and compilation environment. Then we ran the ARM, Thumb, and SetMask (Thumb code with `SetMask` instructions) versions of the programs and compared their performance and code size. We describe our implementation, experimental setup, followed by a discussion of the results.

Implementation. The `xscale-elf gcc version 3.04` compiler used was built to create a version that supports generation of ARM and Thumb code. The above compiler already makes use of higher order registers as spill locations. In other words when no lower order registers are available, one is freed by spilling its contents into a higher order register that is free. Only if no registers are available values are spilled to memory. Therefore the consequence of limited access to higher order registers is generation of `MOV` instructions. We identify the points at which values are spilled into higher order registers and modify the code generated at these points so that the `MOV` instructions are eliminated. `SetMask` instructions are introduced instead. Then we apply the techniques described in this paper to minimize the `SetMask` instructions introduced. By making the register allocator aware of the extra registers made available, it is likely that spill code generated will be reduced. We are currently investigating this, the results shown here provide a lower bound on the performance improvement one can get using our approach. We evaluate the impact on performance by studying the effectiveness of our algorithms in eliminating `MOV` instructions as well as total instruction and cycle counts.

Simulation Environment and Benchmarks. A modified version of the Simplescalar-ARM [1] *simulator,* was used for experiments. It simulates the five stage Intel's SA-1 StrongARM pipeline [21]. The I-Cache configuration for this processor are: 16Kb cache size, 32b line size, and 32-way associativity, and miss penalty of 64 cycles (a miss requires going off-chip). The simulator was extended to support both 16-bit and 32-bit modes, the Thumb instruction set and the system call conventions followed in the `newlib`

c library. This is a lightweight C library used on embedded platforms that does not provide explicit network, I/O and other functionality typically found in libraries such as `glibc`.

The *benchmarks* used are taken from the `Mediabench` [9] and `Commbench` [13] suites as they are representative of a class of applications important for the embedded domain. The benchmark programs used do not require functionality not present in `newlib`. A brief description of the benchmarks is given in Table 1. Code size being a critical constraint, all programs were compiled at -O2 level of optimization, since at higher levels code size increasing optimizations such as function inlining and loop unrolling are enabled.

Table 1. Benchmark Description

Name	Description
`rtr`	Routing Lookup Algorithm
`adpcm`	Adaptive Differential pulse code modulation
`pegwit`	Elliptical Curve Public key Encryption
`frag`	IP packet header fragmentation
`reed`	Reed Solomon Forward Error Correction

Increase in Code Size. Code size is a critical constraint and we show here how our algorithms result in extremely small increases, if at all any, in code size. Figure 15 shows the code size for ARM, Thumb along with the code size by using `SetMask` instructions in the naive way described earlier (*Naive* `SetMask`) and after applying our optimization algorithms (`SetMask`). The increase in code size seen in the naive case has been cut back so dramatically that the `SetMask` instructions have a negligible cost in terms of code size increase.

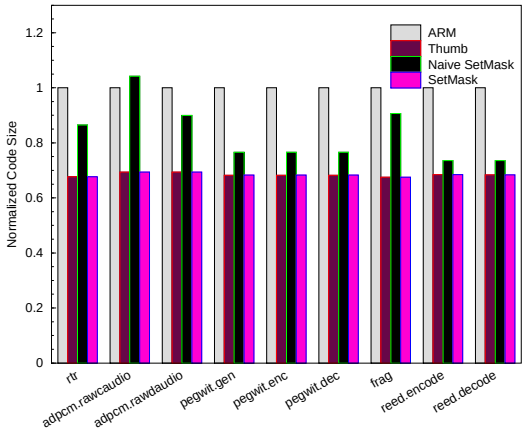

Figure 15. Normalized Code Size

Elimination of MOV Instructions. By using the `SetMask` instruction we effectively cut down the number of `MOV` instructions executed at runtime. Recall that while `MOV` instructions have a single cycle execution cost, the `SetMask` instruction is coalesced with the preceding Thumb instruction using the Dynamic Coalescing Framework, hence having an execution cost of zero cycles. We measured the percentage of executed `MOV` instructions eliminated by making use of our techniques. The results are given in Table 2. As we can see, a significant percentage of `MOV`s (11.7%) introduced by the `gcc` compiler are eliminated by using `SetMask` instructions.

Table 2. Percentage of Executed MOVs Eliminated.

Program	MOVs Eliminated
rtr	21.1%
adpcm.rawcaudio	26.8%
adpcm.rawdaudio	0%
pegwit.gen	6.5%
pegwit.enc	27.6%
pegwit.dec	4.9%
frag	2.2%
reed.encode	10.1%
reed.decode	6.2%
Average	11.7%

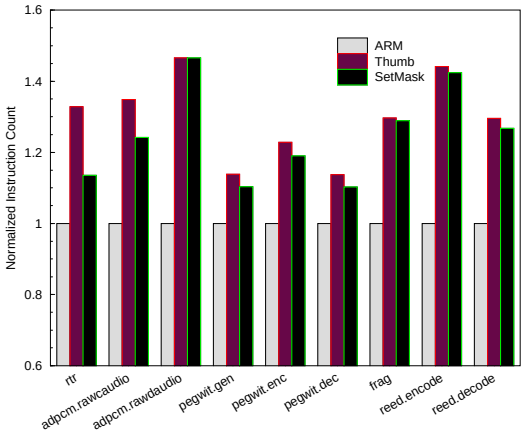

Figure 16. Normalized Instruction Counts.

We also measured the impact of eliminating `MOV`s on total instruction and cycle counts for the programs. Figure 16 shows the dynamic instruction count for ARM, Thumb and SetMask code. We achieve reduction of 0-19% in dynamic instruction count compared to Thumb code. `rtr` gives the the best result of improvement of 19% with other benchmarks giving moderate improvements and `adpcm.rawdaudio` giving no improvement over Thumb code. Figure 17 gives the cycle counts for ARM, Thumb and SetMask code. We achieve between 0-20% speedup in execution time in comparison to Thumb code. In some cases Thumb code is very close, sometimes faster, than the ARM code. This is due to the good cache behavior of Thumb code. `rtr` is a case where although the Thumb code

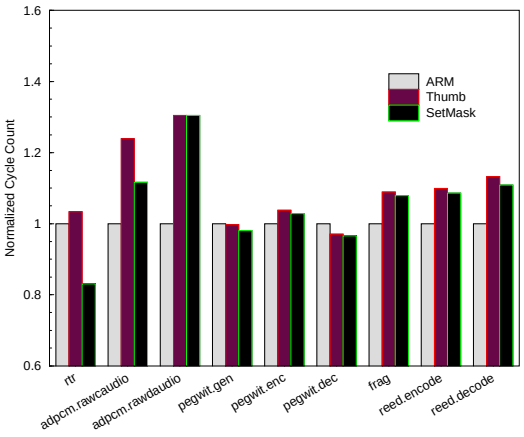

Figure 17. Normalized Cycle Counts.

is not faster than the ARM code, the SetMask code is much faster. Overall, the execution characteristics of SetMask are better than Thumb and comparable to ARM.

In summary, we have shown how with the effective use of `SetMask` instructions one can maintain the code size offered by Thumb code and achieve performance improvements at the same time.

5 Discussion

We have shown our approach on the ARM/Thumb platform with 16/8 registers respectively. Does this approach work for a larger register file? Our algorithms are applied post register allocation and register assignment. How does register assignment affect the `SetMask` instruction insertion? We address these questions here.

Scalability The SetMask mechanism relies on a bitmask which is indexed during the register file access and a mechanism to set this bitmask. When there is a notion of pairs of registers, like in our case, we can use a bitmask to activate different subsets of registers by toggling one bit for each pair. We had 8 addressable registers and 8 corresponding high registers and 8 bits of state corresponding to the 8 pairs. When we scale to a larger register file, we end up with many more non-addressable registers. In this case, we can no longer use one bit of state. We employ multiple bits of state. For instance, if we scaled to 32 registers with 8 addressable registers, we now have sets of 4 registers rather than pairs. Hence we use 2 bits of state rather than 1. We would need 2 `SetMask` instructions to set the 2 bitmasks. Hence the approach scales as long as we have 16-bit instruction encodings for the `SetMask` instructions.

Register Assignment The need for `SetMask` instructions arises when a register which is not in the current active subset is used. A different register assignment clearly changes the placement points for `SetMask` instructions. Hence one could decrease the number of `SetMask` instructions introduced by changing the register assignment to minimize the number of switches between pairs of registers. This could precede our algorithms to minimize the number of `SetMask` instructions. However, as we have seen from our experiments, even without this preceding phase, our algorithms are able to keep the increased code size to a negligible amount. Hence while a different register assignment could precede our algorithms, we did not find the need for it in our experiments.

6 Related Work

Prior work has studied the use of extra registers for high performance processors in various contexts. A 2-level hierarchical register file has been proposed in [14]. This design provides a small first level register file and larger second level register file enabling a larger register file with a larger number of ports. The first level register file has lower access latency compared to the second level register file allowing software pipelined loops to be executed more efficiently. Register Connection [5] has been proposed for superscalars to make more registers accessible to the compiler. A level of indirection is used to connect logical registers to the physical registers. Special register connect instructions are provided that can make changes to this mapping. The ILP available on superscalars allows the performance cost of the register connect instructions to be small. The additional code size introduced by these instructions is significant. While this it not much of a constraint on high performance machines, it is an important concern for embedded processors. Compiler Controlled Memory (CCM) [2] is another technique which tries to reduce the register pressure on the register file. It does so by incorporating a small memory close to the register file. The contents of this memory are managed by the compiler and used to handle spill code. Register windows have been used in the Tensilica Xtensa [19] and SPARC [20] architectures to avoid the saving and restoring of context during procedure calls. In all of the above techniques, the size of the extended register file prohibits their use in embedded processors where power and cost are one of the main constraints.

Recently there have been proposals for the use of extra registers in embedded processors WIMS [12] proposes having several register windows and provides window management instructions which the compiler can use to swap register windows. During register allocation the virtual registers are partitioned into windows using a graph based partitioning algorithm. Code size increase due to window management instructions has not been considered in [12]. Our approach is also more flexible than register windows because it allows various subsets of 8 registers to be active. Differential Register Allocation [15] proposes encoding the register specifier using the difference between consecutive reg-

ister accesses, allowing the compiler to allocate more registers than can be specified using a regular encoding. This scheme, unlike ours, does not provide backward compatibility of binaries. Moreover, it can be used in conjuction with ours – while differential encoding enables more registers to be directly addressed, setmask enables encoding to be changed to address additional registers that cannot be otherwise directly addressed. In [16] a small extended register file is used which is allocated at runtime. The aim is to reduce spill cost by dynamically choosing the extended register file over memory using compiler generated priorities. Offset fields in memory instructions are used to communicate these priorities to the hardware. This approach cannot be used to access existing high registers in Thumb state like our approach.

Prior work has also studied ISA design to allow access to higher registers. [8] proposes shrinking the destination register field of certain instructions and using this extra encoding space for other fields, partitioning the register file based on instruction type. They also describe a register allocation scheme for such an ISA. Mixed width ISAs can be exploited to allow access to both high and low registers by generating binaries with instructions from both the 32-bit and 16-bit instruction sets [7, 4]. There have been several extensions to the ARM architecture [11, 17] that seek to improve performance by allowing access the higher registers. Thumb-2 [11] provides new 16-bit and 32-bit instructions in Thumb state. NEON [17] is a SIMD extension to the ARM architecture that allows access to a special registers file for SIMD instructions. While our goal in this paper was to attack the global inefficiency of Thumb code, the SetMask mechanism can be implemented along with these proposals as it is orthogonal to these techniques.

7 Conclusions

Dual instruction width processors such as the ARM provide two instructions sets - a 16-bit ISA and a 32-bit ISA - with a tradeoff between code size and performance. We have shown how one need not sacrifice code size in order to achieve better performance. In this paper, we have showed how one can address the global inefficiency in Thumb code due to a lack of visibility of high registers. We showed how one can expose the high registers with the help of a new AX instruction - `SetMask`. We also described how one can effectively use these `SetMask` instructions to retain the code size of the original Thumb code and simultaneously achieve performance improvement.

References

[1] Burger, D. and Austin, T., "The simplescalar toolset," *Technical Report CS-TR-96-1308, University of Wisconsin-Madison*, 1996.

[2] Cooper, K. and Harvey, T., "Compiler-controlled memory," *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2-11, 1998.

[3] Furber, S., *ARM System Architecture*, Addison-Wesley, 1996.

[4] Halambi, A., Shrivastava, A., Biswas, P., Dutt, N., and Nicolau, A., "An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs," *Proc. of the Conference on Design, Automation and Test in Europe*, IEEE CS, Washington, DC, 2002.

[5] Kiyohara, T., Mahlke, S., Chen, W., Bringmann, R., Hank, R., Anik, S., and Hwu, W-M., "Register connection: a new approach to adding registers into instruction set architectures," *Proc. of the 20th International Symposium on Computer Architecture*, pages 247-256, 1993.

[6] Krishnaswamy, A. and Gupta, R., "Dynamic coalescing for 16-bit instructions," *ACM Transactions on Embedded Computing Systems*, Vol. 4. No. 1, pages 3-37, 2005.

[7] Krishnaswamy, A. and Gupta, R., "Profile guided selection of arm and thumb instructions," *Proc. of the ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 55-64, 2002.

[8] Kwon, Y-J., Ma, X., and Lee, H.J., "PARE: instruction set architecture for efficient code size reduction," *Electronics Letters*, pages 2098-2099, 1999.

[9] Lee, C., Potkonjak, M., and Mangione-Smith, W., "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330-335, 1997.

[10] Lee, S., Lee, J., Min, S. L., Hiser, J., and Davidson, J. W., "Code generation for a dual instruction set processor based on selective code transformation," *Proc. of the 7th International Workshop on Software and Compilers for Embedded Systems*. Vienna, Austria, LNCS 2826, pages 33-48, 2003.

[11] Phelan, R. "Improving ARM Code Density And Performance," 2003.

[12] Ravindran, R.A., Senger, R., Marsman, E.D., Dasika, G.S., Guthaus, M.R., Mahlke, S.A., and Brown, R.B., "Partitioning Variables across Multiple Register Windows to Reduce Spill Code in a Low-power Processor", *IEEE Transactions on Computers*, Vol. 54, No. 8, Aug. 2005, pp. 998-1012.

[13] Wolf, T. and Franklin, M., "Commbench - a telecommunications benchmark for network processors," *Proc. of the International Symposium on Performance Analysis of Systems and Software*. IEEE, pages 154-162, 2000.

[14] Zalamea, J., Llosa, J., Ayguad, E., and Valero, M., "Two-level hierarchical register file organization for VLIW processors" *Proc. of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 137-146. ACM Press, 2000.

[15] Zhuang, X. and Pande, S., "Differential register allocation," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168-179, Chicago, IL, USA,2005.

[16] Zhuang, X., Zhang, T., and Pande., S., "Hardware-managed register allocation for embedded processors," *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 192-201. ACM Press, 2004.

[17] ARM Inc, *ARM NEON Technical Data Sheet*, 2004.

[18] Intel, *The intel xscale microarchitecture technical summary*, ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[19] Tensilica Inc., *Xtensa Architecture and Performance*, Sep 2002. http://www.tensilica.com/xtensa_arch_white_paper.pdf.

[20] SPARC International Inc., *The SPARC Architecture Manual, Version 8*, 1992. http://www.sparc.com/standards/V8.pdf.

[21] Intel, *Sa-110 microprocessor technical reference manual*, ftp://download.intel.com/design/strong/applnots/27819401.pdf, 2000.

[22] Intel, "The intel pxa250 applications processor," 2002.