Facilitating Compiler Optimizations through the Dynamic Mapping of Alternate Register Structures

Chris Zimmer, Stephen Hines, Prasad Kulkarni, Gary Tyson, David Whalley
Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
{zimmer,hines,kulkarni,tyson,whalley}@cs.fsu.edu

ABSTRACT

Aggressive compiler optimizations such as software pipelining and loop invariant code motion can significantly improve application performance, but these transformations often require the use of several additional registers to hold data values across one or more loop iterations. Compilers that target embedded systems may often have difficulty exploiting these optimizations since many embedded systems typically do not have as many general purpose registers available. Alternate register structures like register queues can be used to facilitate the application of these optimizations due to common reference patterns. In this paper, we propose a microarchitectural technique that permits these alternate register structures to be efficiently mapped into a given processor architecture and automatically exploited by an optimizing compiler. We show that this minimally invasive technique can be used to facilitate the application of software pipelining and loop invariant code motion for a variety of embedded benchmarks. This leads to performance improvements for the embedded processor, as well as new opportunities for further aggressive optimization of embedded systems software due to a significant decrease in the register pressure of tight loops.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; C.1 [**Computer Systems Organization**]: Processor Architectures

General Terms

Experimentation, Performance

Keywords

Register Queues, Software Pipelining, Compiler Optimizations

1. INTRODUCTION

As a consequence of the proliferation of computers in the embedded domains, and the rising complexity of embedded applications, designing customized embedded processors is becoming increasingly difficult. Conversely, the design time for embedded processors, driven by market dynamics, is staying the same, or is in fact decreasing in many cases. One method of reducing the processor design time in embedded domains is to use general-purpose solutions for the design of the processor. These conventional embedded processors provide several different designs, and also support extensions to customize the processor for particular applications. The ARM processor is a prime example of such a customizable embedded processor. The ARM architecture supports several extensions such as Thumb [10] and Thumb2 [5], each of which is designed to support additional instruction set enhancements. As embedded software complexity increases, the need for highly-tuned compilers with sophisticated optimizations becomes more apparent.

The ARM is a 32-bit RISC processor architecture with 16 registers, only 12 of which are typically available for general use. Even simple tasks can require sequences of instructions that exhaust the supply of free registers, leading to poor overall performance. Embedded system constraints can often be a limiting factor for many aggressive compiler optimizations that target improved machine performance. Techniques like software pipelining [4] require additional registers to be effective in eliminating loop inefficiencies.

Dynamic mapping of alternate register structures help to alleviate the shortage of registers by using small additional register structures to exploit register usage patterns found in code or produced by compiler transformations. By identifying these exploitable patterns, a processor could utilize its alternate register structures to act as an enlarged register mechanism which is accessible through the architected register file. This would allow the compiler greater flexibility during software optimizations and greater control over the available registers during register allocation. The overall impact of this approach can lead to significantly improved performance and a reduction in application register pressure as well as enable more aggressive compiler optimizations to be performed in areas otherwise impossible. In this paper we propose a new architectural feature that would more often enable aggressive compiler optimizations such as software pipelining and loop invariant code motion on general purpose embedded processors. These features will provide a non-invasive mechanism to reduce register pressure while exploiting common register usage patterns within applications. These new features enable the compiler to be successful in applying these optimizations, leading to improved application performance and reduced register pressure.

2. SOFTWARE PIPELINING

Software pipelining [4] [7] is an optimization which attempts to extract iterations from a loop enabling high latency operations to be

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

```
        int A[1000], B[1000], C[1000];
        void vmul() {
          int I;
          for (I=999; I>= 0; I--)
            C[I] = A[I] * B[I];
        }
Prologue:
 -ldr r6, [r2, r5, #2]  ;  load A[999]
  ldr r7, [r3, r5, #2]  ;  load B[999]
  sub r5, r5, 1
 -ldr r17, [r2, r5, #2] ;  load A[998]
  ldr r19, [r3, r5, #2] ;  load B[998]
  sub r5, r5, 1
 -ldr r18, [r2, r5, #2] ;  load A[997]
  ldr r20, [r3, r5, #2] ;  load B[997]
  mul r8, r6, r7        ;  A[999]*B[999]
  mov r6, r17
  mov r17, r18
  mov r7, r19
  mov r19, r20
  sub r5, r5, 1
Loop:
  ldr r18, [r2, r5, #2] ;  load A[I]
  ldr r20, [r3, r5, #2] ;  load B[I]
  mul r21, r6, r7       ;  A[I+2]*B[I+2]
  str r8, [r4, r5, #2]  ;  store C[I+3]
  mov r6, r17
  mov r17, r18
  mov r7, r19
  mov r19, r20
  mov r8, r21
  sub r5, r5, 1
  bnz Loop
...
```

Figure 1: Software pipelined vector multiply showing the extracted loop iterations

computed early in order to remove stalls. Software pipelining was first employed in VLIW [3] and decoupled architectures [11]. This optimization relies on the availability of extra registers to hold the values of the loop-carried dependencies across iterations. Finding enough available registers in conventional embedded processors is difficult even for extremely simple loops. Figure 1 shows an example of a vector multiply with three iterations of the loop extracted to the prologue. These extracted iterations allow the higher latency operations such as the multiply to start its calculation early avoiding pipeline stalls incurred by that instruction from the loop.

Two methods that support the register renaming requirements of software pipelining are modulo variable expansion (MVE) [4] and the rotating register file (RR) [8, 9]. Modulo variable expansion is an approach that uses a software based mechanism which renames each concurrent live variable and provides it with its own name. Modulo variable expansion increases both the architected register requirements as well as the eventual size of the pipelined loop to handle the renaming that has occurred in the loop. A rotating register file is a hardware based register renaming mechanism which dynamically renames the registers for each instance of a loop variable. Rotating registers provide the ability to access a variable that was defined several iterations prior to its use. This feature provides this ability by creating a level of indirection to the register specifier which correlates loop iteration count. Rotating registers reduce the eventual size of the pipelined loop, however they require a significant amount of architected registers to generate a schedule.

Both of these methods provide techniques which maintain the register requirements that software pipelining creates. However with software pipelining, as the functional unit latencies grow, so do the register requirements needed to apply this optimization. A single long latency operation in a loop will generate a need for more interleaved instances of the loop in order to successfully pipeline it. When this situation occurs, the number of registers available in a

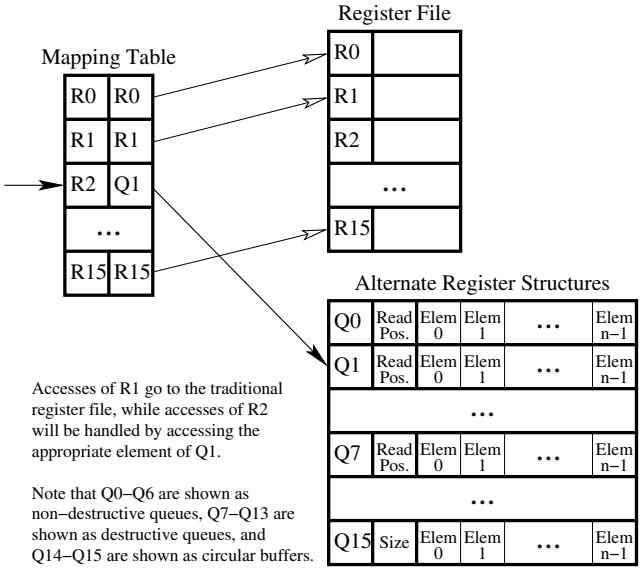


Figure 2: Register file access

general-purpose embedded processor will often be insufficient to apply the optimization. One possibility would be to increase the number of architected registers. This would require a both a modification of the architecture and a rewrite of the entire ISA to support the encoding of the new registers. Our method enables software pipelining for a current general-purpose embedded architecture utilizing its preexisting ISA and requiring only minimal modifications of the hardware.

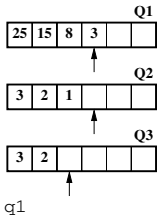
3. MAPPING OF ALTERNATE REGISTER STRUCTURES

Conceptually we wish to develop a system which enables aggressive optimizations in reduced register environments. Our method is to create a new set of registers within the architecture that would significantly increase the amount of available register space while still remaining virtually invisible to the ISA. To achieve this goal we applied the use of new register types that mimic the behaviors of queues to store many data values yet still remain visible through a single architected register reference.

The first component designed to enable the application configurable extension was the value mapping table. The concept of the mapping table in our work is very similar to the rename table that can be found in many out-of-order processor implementations. Using a similar concept to the ideas in the Register Connection approach [3], this table provides a register lookup and corresponding data structure from which to obtain the data. This structure is a very quick access map table and is the only redirection required to implement this system. This map table is modified and set as the first logic that must occur during every access to the architected register file. The size of the table in our design was limited to the number of registers available in the architected register file, but could even be made smaller and faster to access by removing special purpose registers like the program counter and stack pointer.

The implementation of this map is such that the lookup is based on the address of the register itself. The table will then provide a value indicating from which register file the value must be retrieved. If there is no value set in the table, then the value will be retrieved from the specified register in the architected register file.

```
int A[1000], B[1000], C[1000];
void vmul() {
  int I;
  for (I=999; I>= 0; I--)
    C[I] = A[I] * B[I];
}
```




```
qmap r6, #4, q1
qmap r7, #4, q2
qmap r8, #2, q3
Prologue:
  ...  ;  6 loads and 2 mults
Loop:
  ldr r6, [r2, r5, #2] ;  load A[I]
  ldr r7, [r3, r5, #2] ;  load B[I]
  mul r8, r6, r7       ;  A[I+1]*B[I+1]
  str r8, [r4, r5, #2] ;  store C[I+3]
  sub r5, r5, 1
  bnz Loop
Epilogue:
  ...  ;  1 mult and 3 stores
```

Figure 3: Registers being mapped into queues

Our alternate register files can be defined as a fast access memory structure that is large enough to hold multiple values, but enforces a reading and writing semantic to all of the values in it. The reading and writing semantic is used to exploit identifiable reference patterns that can be found in code or caused by the process of optimization. The cost of these additional registers in the hardware would be similar to the cost of the architected register file in terms of area and power. We have implemented several different alternate register files which mimic these behaviors. Figure 2 shows the modifications to an access to the architected register file during either the decode or writeback stages of the pipeline. The map check occurs anytime there is a register file access. From this figure it can be determined that there is no mapping between R1 and any of the alternate register structures. If the value contained in register R1 was needed in the next stage of the pipeline, the map table looks up the mapping and the processor accesses the value from the architected register file. This table shows a mapping between R2 and Q1. This means that any set or use request for R2 will instead be accessing the alternate register structure Q1. The register queue enables a single specifier to store significantly more values than a conventional register file.

Register queues are the primary type of the alternate register file that we have used in this study. The FIFO reference behavior is an identifiable pattern frequently found in the code of many applications and the primary reference pattern constructed in the use of software pipelining. Register queues in our terms refer to an alternate register file that can hold multiple data items at once using a single FIFO register and when the structure is referenced the oldest data item is the value that is returned. In our implementation of register queues we found it necessary to create different semantics for certain types of queues that allow us greater flexibility when using them in different scopes. Queues can use destructive or non-destructive read semantics, which we examined in our research.

Circular buffers are another type of alternate register structure used to hold a set of values referenced in a repeated manner. In many loops that occur in embedded programs the compiler often finds it helpful to use registers to hold loop-invariant values. Loops can often use several different registers to store these values. A circular buffer can be used to store all these values. The pattern of references can be loaded into the circular buffer prior to entering the loop. When a read occurs in the register mapped to the circular buffer, the value will be retrieved from the head of the buffer and then the read position will be incremented to read the next value. When the read position has reached the end it will loop back to the beginning of the structure and begin providing the value from that point. Circular buffer register files are a successful mechanism for storing all of the loop invariant values and providing correct data throughout the loop.

The modifications to the instruction set architecture were designed so that no changes to the registers or instructions themselves were made. We require only one addition to the existing ISA: an instruction which controls both the mapping and unmapping of an alternate register structure. To accomplish this task, it is assumed that the semantics of the instruction can take on different meanings depending on what structure it is referencing. We refer to this instruction as **qmap**. This instruction contains three pieces of information and performs all of its tasks in the decode stage of the pipeline. The qmap instruction is laid out as follows:

qmap *register specifier*, *mapping semantic*, *register structure*

- *register specifier* - The register specifier refers to the register from the architected register file which will point to an alternate register structure.
- *mapping semantic* - The mapping semantic refers to the set up information for the alternate register structure. In the case of non-destructive read queues, this sets the position at the end of the queue.
- *register structure* - The register structure specifier itself is numbered similarly to the architected register file.

An application of this system is demonstrated in Figure 3. In this example the vector multiply from Figure 1 is modulo scheduled and the latency is spread over three extracted iterations. The three qmap instructions map three different loop carried dependencies into queues.

4. ENHANCING OPTIMIZATIONS

Our proposed application configurable processor is an extension to a conventional embedded processor that enables the configuration of the register file to support the reference patterns of an application. A register queue is a register file structure that can take advantage of FIFO behavior within an application. This behavior is exploited by allowing multiple outstanding writes to the same register, while retaining all the values written before the reads occur. This structure allows a single register specifier to be associated with multiple live values. Therefore, a single register queue can replace the sets and uses of many different registers with a single register mapped into one of these structures. This ability is useful because quite a few important code optimizations often force register accesses into a FIFO pattern, and traditionally rely on extra registers to be able to avoid spilling values to memory.

Our method of software pipelining uses iterative module scheduling [6], a method in which the loop instructions are reordered based on dependencies and hardware restrictions to create the pipelinable loop. The algorithm we applied first identifies suitable loops for software pipelining that match specific criteria. Two renaming phases were used in determining which live ranges would be mapped into a register queue. The first renaming sequence was performed prior to modulo scheduling. A backup copy of the loop was

```
foreach innermost loop do
    create priority graph
    identify obvious queue candidates
    modulo schedule based on priority
    identify generate queue candidates
    generate prologue code
    generate epilogue code
    find available registers for mapping
    foreach mapping that requires a register do
        while conflicts do
            identify and resolve possible register conflicts
            associate an available register with mapping
    resolve memory offset conflicts
    insert mapping code into prologue and epilogue
    if successful then
        write generated software pipelined code over old loop
    else clear generated code
```

Figure 4: Queue identification algorithm

created and any instructions that had a latency greater than the iteration interval of the loop would have its live range renamed to our new register type. This new register type would disable the compiler's register analysis optimizations from incorrectly identifying our modified RTLs as dead and removing them. The second renaming occurs after scheduling has been applied and the rest of the newly formed loop carried dependencies could be identified to be promoted to a queue. The next step uses iteration calculations that were determined during loop scheduling to generate a prologue and epilogue for the newly formed kernel. Having previously replaced the loop carried live ranges with our own custom register type; we are then able to identify registers to map into queues to contain the reference behaviors. The pseudocode for applying registers queues to modulo scheduling is provided in Figure 4.

5. RESULTS

Using the SimpleScalar [2] simulation environment we were able to conduct our experiments using several different processor setups with varying latencies. The results were collected using the cycle-accurate simulator *sim-outorder* configured to the specifications of the ARM processor. We used the VPO [1] compiler backend ported for the ARM ISA. The compiler had to be modified to be able to support the use of different register types. Our preliminary tests were performed on a simple in-order ARM processor with added feature support for our alternate register structures.

We obtained our first group of results using several DSP benchmark kernels several of which from the DSPstone [15] suite. We measured the steady state performance of each loop. Figure 5 depicts the percent difference in performance for software pipelining with register queues versus the base loop which could not be pipelined without queues because of register pressure. Our preliminary results shows in Figure 5 that as the latency grows for multiplies, we are able to successfully pipeline the benchmark loops and realize up to a 60% improvement in performance. The increase in the multiply latency is a realizable factor in low power embedded design. Many low power multipliers trade off extra cycles in order to improve power. These improvements do come at the cost of increased code size of the loop up to roughly 300% in some cases, which is due to the prologue and epilogue code needed by software pipelining to align the loop iterations. Figure 6 shows the performance savings as the load latencies rise. These loops often provide more high latency instructions to schedule out of the loop. In many of the lower latency tests, iterative modulo scheduling was able to generate a loop that did not need a prologue or epilogue. In many

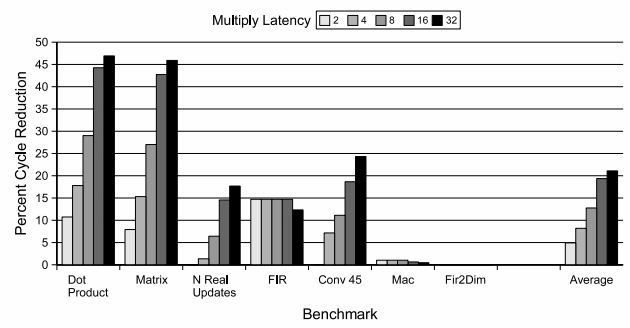


Figure 5: Scaling multiply latency

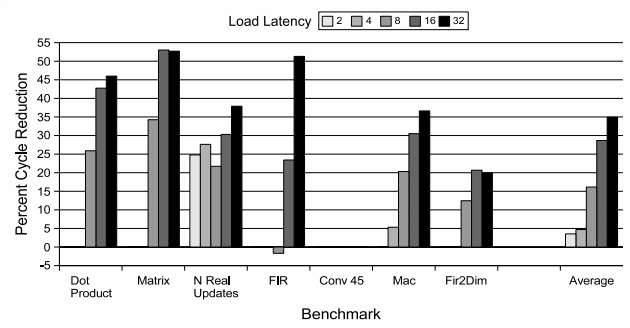


Figure 6: Scaling load latency

of our benchmarks we found that by applying software pipelining with register queues we are able to circumvent increasing register pressure in many simple cases by as much as 50 % for the ARM. This means that software pipelining would require 50% of the usable registers for the ARM in order to even be applied.

The performance loss in Fir (with a multiply latency of 32) in Figure 5 is due to the scheduling of a loop carried inter-instruction dependency. Table 1 shows the relationship between the original number of registers found in a few of the loops which were software pipelined and the number of registers needed to pipeline the loops using our alternate register structures. The final column in the table shows the number of registers which the alternate register structures consumed. The middle row of the table shows the number of registers needed for software pipelining when used in conjunction with register queues. The total number of registers needed to pipeline each of these loops is the summation of the second and third column for each row.

6. RELATED WORK

There have been several approaches for reducing the register pressure caused by software pipelining. These methods for reducing register pressure work under similar constraints as with our register queues, however register queues offer much more flexibility without the cost of significantly modifying the ISA.

The register connection approach introduced the idea of adding an extended set of registers to an ISA as a method of reducing register pressure in machines with a limited number of registers. This method employed a mapping table to associate a register in the register file with one of the extended registers. The map table used a one to one mapping for each register in the extended set. The register connection approach worked well for remapping scalars and various other data, but the overhead of mapping became expensive when using arrays and other large data structures.

Table 1: Architected Register Utilization

Benchmark	Original	After SWP	Mapped
Loads 8x4 Register Savings Using Register Structures			
N Real Updates	10	10	6
Dot Product	9	9	4
Matrix Multiply	9	9	4
Fir	6	6	4
Mac	10	8	10
Fir2Dim	10	10	4
Loads 16x4 Register Savings Using Register Structures			
N Real Updates	10	10	6
Dot Product	9	9	4
Matrix Multiply	9	9	4
Fir	6	6	4
Mac	10	8	12
Fir2Dim	10	10	4
Loads 32x4 Register Savings Using Register Structures			
N Real Updates	10	10	9
Dot Product	9	9	8
Matrix Multiply	9	9	8
Fir	6	6	12
Mac	10	8	18
Fir2Dim	10	10	8

Register queues [12] is the approach that is most similar to ours. Using register queues to exploit reference behaviors found in software pipelining showed that this method is effective in aiding the application of these optimizations. Exploiting the FIFO reference behavior that is caused by software pipelining, register queues was an effective means of holding the extra values across iterations and this significantly reduced the need to rename registers. However, this method limits the types of loops that can be effectively software pipelined because of constraints set by the reference behavior of the queues themselves. Our method described in this paper is an automation of this system with the addition of several tools which aid us in employing register queues to software pipelined loops.

Rotating registers [8, 9] is an architectural approach for more effectively using registers to hold loop carried values than simple register renaming. A single register specifier can represent a bank of registers which will act as the rotating register base. Use of rotating registers is similar to the renaming that would typically occur in software, but instead is all accomplished in the hardware. This method requires that each of the registers in the rotating bank be an accessible register, which in a conventional embedded architecture would require a larger specifier for a register that may not be possible in the given ISA. Application configurable processors provide much of the flexibility of the rotating register file, with only a small added cost for each access.

The WM machine [14, 13] is a completely different concept of the traditional machine that utilizes FIFO queues that operate independently and asynchronously to manage the many different aspects of the traditional pipeline. This system is designed as a series of connected queues that manage the different functions of the pipeline. This paper introduced the concept of using queues in place of registers as a quick storage mechanism.

7. CONCLUSIONS

Our work has shown that using a dynamic mapping of alternate register structures can greatly reduce the register restrictions that inhibit many compiler optimizations in embedded systems. This enables much more agressive optimizations to be performed on the loop kernels that make up a majority of the execution time in embedded applications. Register mapping support can offer to data storage what extensible instruction sets offer to ALU operation; in many cases the application of a customized structure can improve execution efficiency with little or no increase in microarchitectural complexity. This approach to register file organization has been developed with only minor modifications to the instruction set and can be employed on almost any existing processor design. Our research has also shown that it is possible to modify compiler optimizations to automate the allocation and modification of these alternate register structures to make existing optimizations more effective. Our future work will focus on different alternate register files to exploit other identifiable reference behaviors caused by code optimizations and to aid the compiler in identifying these situations for the optimizations which are available.

Acknowledgments

We thank the anonymous reviewers for their constructive suggestions. This research was supported in part by NSF grants CCR-0312493, CCF-0444207, and CNS-0615085.

8. REFERENCES

[1] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin - Madison, Computer Science Dept., June 1997.

[3] M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered vliw architectures. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 386, Washington, DC, USA, 1998. IEEE Computer Society.

[4] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.

[5] R. Phelan. Improving ARM code density and performance. Technical report, ARM Limited, 2003.

[6] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[7] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, pages 131–139, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[8] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 283–299, New York, NY, USA, 1992.

[9] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towie. The cydra 5 departmental supercomputer. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[10] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.

[11] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[12] G. S. Tyson, M. Smelyanskiy, and E. S. Davidson. Evaluating the use of register queues in software pipelined loops. *IEEE Trans. Comput.*, 50(8):769–783, 2001.

[13] W. A. Wulf. The wm computer architecture. *SIGARCH Comput. Archit. News*, 16(1):70–84, 1988.

[14] W. A. Wulf. Evaluation of the wm architecture. *SIGARCH Comput. Archit. News*, 20(2):382–390, 1992.

[15] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and multimedia applications – an evaluation using DSPstone. In *ICSPAT: International Conference on Signal Processing Applications and Technology*, November 1995.