On Memory Behavior of Scalars in Embedded Multimedia Systems

Osman S. Unsal[1], Zhenlin Wang[2], Israel Koren[1], C. Mani Krishna[1], Csaba Andras Moritz[1]
The Departments of Electrical and Computer Engineering[1] and Computer Science[2]
University of Massachusetts, Amherst, MA 01003

Abstract

In an earlier paper about the FlexCache project [18], we described our vision of a multipartitioned cache where memory accesses are separated based on their static predictability and memory footprint, and managed with various compiler controlled techniques supported by instruction set architecture extensions, or with traditional hardware control.

In line with that vision, this paper describes our work in progress related to the memory performance and memory management of scalars. Our focus in this paper is embedded multimedia architectures, but the methodology described can be applied to other classes of applications.

In particular, we establish the minimum size of a memory partition that would allow us to map and manage all scalar accesses in a program statically, and describe compiler techniques to automate the extraction of this information. Additionally, we study the cache behavior of scalar accesses for these architectures, including reduction in cache misses due to separation of scalars from other types of memory accesses. Finally, we evaluate the impact of register file size on the volume of scalar related memory accesses, and its impact on the applications' overall cache performance.

1 Introduction and Motivation

The recent proliferation of palmtops, MP3 players and internet-enabled wireless phones has ignited interest in embedded multimedia systems. These systems have to be fast and energy efficient. As such, they have tight memory/processing requirements. Therefore, understanding their memory/caching behavior is of paramount importance.

In an earlier paper about the FlexCache project [18], we described our vision of a multipartitioned cache where memory accesses are separated based on their static predictability and memory footprint, and managed with various compiler controlled techniques. This paper addresses the cache behavior of scalar accesses and its memory footprint in order to enable a fully static memory management in a logical memory partition.

Although prior studies into memory behavior of arrays for embedded systems have been conducted, the study of the memory footprint of scalars has lagged behind. Here we report our ongoing work in closing this gap. This paper presents techniques and results for scalar memory accesses in embedded multimedia systems. Our preliminary results show promise and we hope that this work will heighten interest in this area.

The research spans compiler and architectural domains. Our contributions in this paper are threefold:

- First, we experimentally establish the memory size requirements of scalars for embedded systems running media applications. We present a new compiler algorithm to automatically extract this information, as would be required in a multi-partitioned cache.
- Second, by separating scalar accesses from array accesses, we expect decreased cache interference and improved static predictability. This aspect is especially important for hard real-time embedded systems.
- Third, we report the effect of register file size on scalar memory accesses. Register file sizes of various existing embedded CPUs are used for this study. We also analyze the impact of future generation embedded CPU register file sizes on scalars.

The rest of this paper is organized as follows. In Section 2, we provide a brief literature survey and reiterate our motivation. Section 3 describes the experimental setup. Section 4 provides the results and in Section 5 we conclude with a brief summary and a synopsis of future work.

2 Previous Work

This work builds upon the framework in [17, 18]. Previous memory behavior research effort primarily targeted array structures [11, 20]. On the other hand, architectural support to improve memory behavior include split caches which were discussed in [16]. Albonesi [2] proposed selective cache ways, a vertical cache partitioning scheme. Panda et al. [21] proposed a scratchpad memory based partitioning. Mueller [19] sketched some broad ideas on compiler support for cache partitioning. Region based caching for embedded processors was analyzed in the context of reducing power dissipation rather than performance in [13]. Combined compiler/architectural efforts toward increasing cache locality [15] have also exclusively focused on arrays. For multimedia systems one previous work has considered reconfigurable caches [22], using the recently introduced Mediabench benchmark in the performance analysis, with comments on compiler controlled memory. Burlin [7] concentrates on optimizing stack frame layout in embedded systems. Cooper and Harvey [8] look at compiler-controlled memory. Their analysis includes spill memory requirements for some Spec '89 and Spec '95 applications. Engblom [9] and Lee et al. [12] discuss why Spec is not a suitable benchmark for embedded systems.

The above research, although preoccupied primarily with memory behavior of arrays, provided valuable pointers for our work. In this paper we consider scalar memory accesses, not only array or spill memory accesses, and we target embedded systems running a suite of media applications. We develop a compiler heuristic to calculate the memory requirements of scalars and discuss the impact of architectural design choices of embedded systems on scalars.

3 Experimental Setup

We use the recently developed Mediabench benchmarks [12] in our experiments. Mediabench is a collection of popular embedded applications for communications and multimedia. We chose Mediabench, since other benchmarks such as SPECint, DSPstone or Dhrystone are not suitable for embedded systems [5, 9].

Figure 1 contains a block diagram of our framework. We needed a detailed compiler framework that would give us sufficient feedback, is easy to understand, and allow us to change the source code for our modifications. With this in mind, we chose the SUIF/Machsuif suite as our compiler framework. SUIF [23] does high-level passes while Machsuif [14] makes machine specific optimizations. Our main focus is Machsuif's register allocator pass, Raga. Raga makes the transition from virtual registers into real registers and does register allocation. The allocation uses a graph coloring heuristic to assign registers to temporaries. We have made modifications to Raga to annotate scalar memory accesses. The resulting annotated assembler code targets the Alpha processor. We have amended the assembler code by inserting NOP instructions around the scalar memory operations, thus *marking* them. The scalar memory accesses consist of spills and register promotion related memory accesses.

We used the Simplescalar tool suite [6] to run the Alpha binaries and collect the results. We have modified Simplescalar to recognize the scalar memory operations in the *marked* code. Our baseline machine model is a single-issue in-order processor. Lee et. al. [13] use an identical Simplescalar configuration in their power dissipation analysis of region-based caches for embedded processors. To determine the baseline cache size, we did a survey of cache sizes of embedded processors. As Table 1 indicates, embedded processor data cache sizes are usually small. Therefore, we have selected a data cache size of 2K for our experiments.

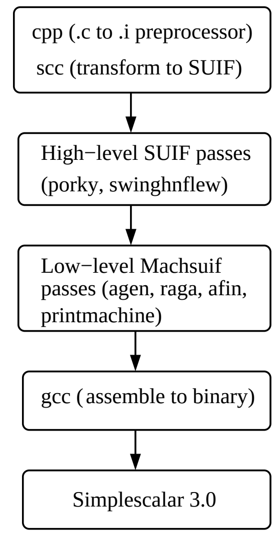

Figure 1: The Experimental Setup Block Diagram

4 Results

4.1 Motivational Example

We start with a motivational example. Consider the sample program in Figure 2. The program consists of x scalar variables being written in a chain-dependent fashion, after which a single array element is written per loop iteration. We define the scalar miss ratio to be the ratio of scalar misses to total misses. Consider the scalar miss ratio for 32 scalars which is 34%. When we increase the number of scalars to 64 the ratio increases to 46%, although the memory footprint of 32 additional scalars is small. This points to the fact that interference between the scalar and array accesses are chiefly responsible for the increase in the scalar miss ratio. Therefore, if we can separate the array accesses from the scalar accesses, this ratio and the overall miss rate will decrease. Next, we present our results based on Mediabench applications for embedded systems.

Processor	Cache Size
Samsung ARM7	2K
SparcLite	2K
PA-RISC HP	1K to 2K
Power PC 403GA	1K
Hitachi SH-II	4K unified
Coldfire 5102	1K
Embedded Pentium	8K
MIPS Jade	1 to 8K
Sandcraft SR-1-GX	8K

Table 1: Data cache sizes for typical embedded CPUs. SRAM scratchpad areas are available in the Samsung ARM7, Hitachi SH2 and Fujitsu Sparclite.

```
main( ) {
    loop {
        scalar_accesses;
        array_access[ ];
        }
    }
```

Number of scalar temporaries

32	64	96
$\frac{13133}{37676}$=0.34	$\frac{24765}{53632}$= 0.46	$\frac{37597}{79597}$=0.47

Figure 2: Scalar misses for the synthetic example. Here the integer array is of size 2048, and the columns denote the number of scalar variables in the example. Scalar operations are of the form: $Variable_{n+1} = Variable_n \mp constant$. There is a single array access per loop iteration. The loop is iterated 100000 times. The cache is 2K-direct mapped.

4.2 Memory Size

We use two yardsticks for experimental evaluation of the scalar memory size requirements of media applications. The first of these is the static memory evaluation. It is static in the sense that the results were extracted by a compile-time analysis of assembler code. We isolated the scalar memory operations in every routine. We then determined the granularity of data by instruction analysis, i.e., the granularity is 8 if the move is a quadword instruction, 2 if it is a word instruction, and so on. We then identified the unique scalar accesses by counting multiple accesses into the same memory location only once and by taking the maximum of the pertaining granularities. The results given in the first column of Table 2 indicate that memory size requirements are modest.

However, the static estimate is pessimistic since not all of the data space is traversed during execution. We therefore, developed a second yardstick, a dynamic memory evaluation which provides a tighter, more robust bound. We recompiled the Mediabench benchmarks to record runtime routine use information. We executed each benchmark with its default input set and extracted the *dynamic* call-tree information by using the *gprof* profiling utility. Then, for every routine we noted the scalar memory requirements as in the static technique. Traversing the tree from the root to each leaf, adding up the unique scalar accesses from each routine, and finding the critical path, i.e., the path with the maximum size requirement, yields the result. We supply the *dynamic* call-tree for the EPIC application in Figure 3 as an example of this process. The memory requirements thus obtained are shown in the second column of Table 2. The results suggest that the memory footprint of scalars in media applications for embedded systems is quite small. These results will guide the choice of our architectural optimization schemes. We next present our compiler technique to automate the scalar memory size estimation.

(In Bytes)	Static	Dynamic
ADPCM	0	0
EPIC	321	203
G721 Encode	48	32
GSM	202	146
JPEG Encode	502	83
MPEG Encode	2125	604
PEGWIT	98	16
RASTA	618	152
PGP	394	358
MESA	2191	770

Table 2: The Memory Size Requirements

Intuitively, the upper bound of the size of the scalar buffer is the maximum of the distinct scalars along all program execution paths. An algorithm that accurately calculates this bound needs inter-procedural analysis and a complex data-flow analysis. Here we present a good approximation. Our algorithm conservatively assumes that the scalars along all paths are distinct. It simply adds the number of bytes needed for each scalar. Of concern here are loops in the control flow graph and recursive calls in the call graph. We can reuse the scalar space for loops and need only count it once. To accomplish this, the algorithm first marks back-edges in the control flow graph which are not going to be traversed. For recursive calls, when there is no register promotion on stack accesses, such as parameters and local variables, the compiler can still ignore the recursion because the scalar buffer can be reused. Otherwise, it will be impossible to compute the upper bound because the depth of recursion is usually unknown at compile time. However, stack accesses for recursive calls usually have no reuse. We can assume that the buffer replacement policy can take care of those scalars. Therefore, in our algorithm, we also count the space for recursive calls only once by ignoring back-edges in the call graph.

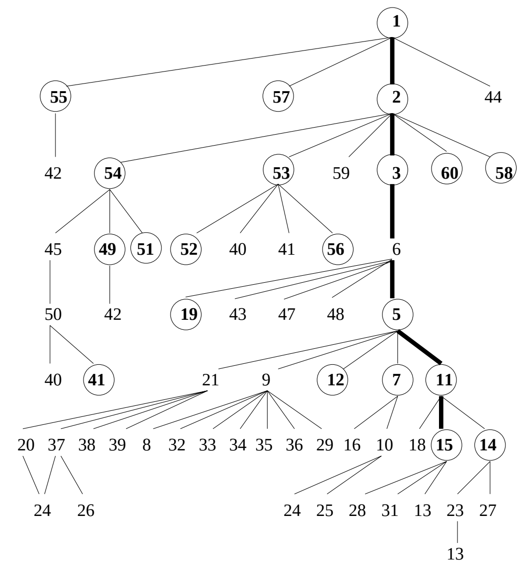

Figure 3: Call-Tree and the Critical Path for the EPIC benchmark. Here the routines are numbered from 1 (the function *main*) to 60. The routines with scalar memory accesses are circled and the path with bold lines is the critical path.

Our algorithm is divided into two phases. The first phase calculates the bound for each routine, ignoring all routine calls. The second phase traverses the call graph to compute an upper bound for the whole program. See Algorithms 1 and 2 (in the Appendix) for the flow of phases 1 and 2, respectively. In these algorithms, we assume there are three attributes for each basic block and call node, *resolved*, *scalarBound*, and *localScalarSize*. The *localScalarSize* of a basic block is the total number of bytes for all scalars in the basic block. The *localScalarSize* of a routine is its scalar buffer upper bound without taking routine calls into account. The *scalarBound* of a basic black is the scalar bound along all simple paths from the entry block to the current block in the control flow graph. The *scalarBound* of a routine is the scalar bound along all simple paths from the *main* routine to the current routine in the call graph. We say that a basic block or a routine is *resolved* when its *scalarBound* is known. Assuming there are N routines in a program and the maximal number of basic blocks of a routine is M, then the complexity of the algorithms is $O(NM^2+N^2)$.

4.3 Register File Size

For memory analysis of arrays, optimizing the cache is more important than the register file architecture, since array accesses seldom use registers. However, for scalars the situation is different. The register file size can have a direct impact on spills and thus impact performance. Here we analyze the impact of register file sizes on scalars. We take a two-step approach: first, we do a survey of the register file sizes for *current* embedded CPUs and use these results to drive our experiment. Second, we gauge the impact of expanded register file sizes in *future* embedded processors.

Table 3 shows the register file sizes on some typical embedded CPUs: the size ranges between 16 and 32 except for the embedded Pentium which has 8 general purpose registers. We therefore, varied the register file size from 16 to 32 in our experiments. We modified Machsuif passes and architectural definitions to output binaries for different register file sizes. Then we noted the static number of scalars inserted into the instruction stream. The results in Figure 4 show that there is a considerable number of scalar memory accesses for 16 registers. Another point is that for some particular benchmarks (e.g. JPEG Encode) the number of scalar memory accesses is more dramatically decreased than others as more registers become available. This is because the register pressure is more unevenly distributed in those benchmarks, i.e., only a few routines exhibit intense register pressure. Once those are relieved through additional registers, the decrease in scalar register spills is more steep.

Usually, the current methods and

Processor	Register File Size
Samsung ARM7	15
SparcLite	32
PA-RISC HP	16+16
Power PC 403GA	32
Hitachi SH-II	16
Coldfire 5102	16
Embedded Pentium	8
MIPS Jade	32
Sandcraft SR-1-GX	32

Table 3: Integer Register File Sizes in Current Embedded CPU's

techniques used in general microprocessors migrate to embedded systems with a couple of years time lag. We believe that the integer register file sizes will follow the same trend. Therefore, we project the embedded CPU integer register file size to grow to 64, 128 and 256. We extended our analysis by modifying Machsuif to output code for larger register file sizes. The results are shown in Figure 5. Note that for large register file sizes all the register spills are eliminated, the only remaining scalar memory operations are register promotion related; this is the reason for the flattening out of the scalar memory accesses. The implication is that, as far as scalars in media applications are concerned,

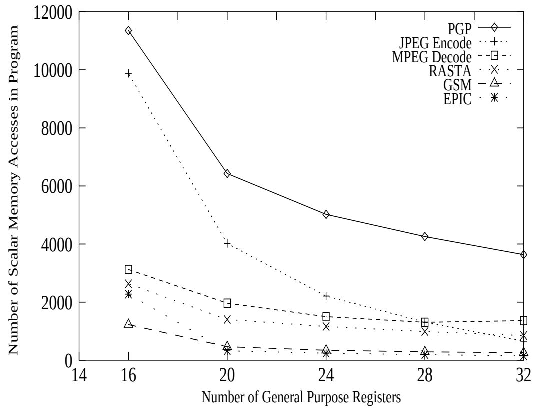

Figure 4: Number of Scalar Memory Accesses With Register File Size

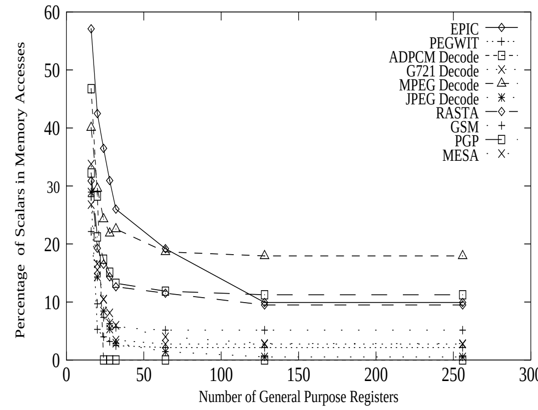

Figure 5: Percentage of Scalar Memory Accesses for Extended Register File Size

increasing the register file size will not bring any additional benefits. This is especially true for register file sizes larger than 64. Thus, we experimentally establish what has been an industry insight with general-purpose CPUs [4]. This provides a guidance to the designers of future embedded/multimedia CPUs: the number of integer general-purpose registers should be at most 64, the additional chip real estate could be devoted to other functional units (e.g., caches) that offer better incremental performance.

Our analysis also includes a cross-architectural comparison as seen in Figure 6. We used an Intel X86-family targeted version of Simplescalar for this analysis. The 8 register X86 has significantly more scalar memory accesses than the 32 register Alpha. This is due to Machsuif's register allocator, Raga. Raga is based on a graph coloring heuristic and as argued in [1], register allocation based on graph coloring is sensitive to the number of registers, in particular when the number of available registers is low. Here, we experimentally verify this argument. *Therefore, compiler designers for embedded CPUs, which typically have fewer registers, should develop new register allocation heuristics.* Work in this direction has already started [1]. We also comment on an important property leading to a dual conclusion. Sometimes, increasing the register file size can increase scalar memory accesses. This may seem counterintuitive at first. However, consider Figure 7 for the MPEG benchmark. As the register size is increased from 28 to 32, the number of scalar memory accesses actually increases. This is due to Raga, which uses a graph-coloring heuristic to assign registers since the problem is NP-complete. The use of this heuristic creates a phenomenon similar to the Belady anomaly in paging [3]. The conclusion that can be drawn: *embedded CPU designers should be aware of the characteristics of their target compiler in choosing their design point.* In summary, the above experiments show that the compiler/architecture coupling in embedded systems is stronger than previously assumed and should be considered at the design phase.

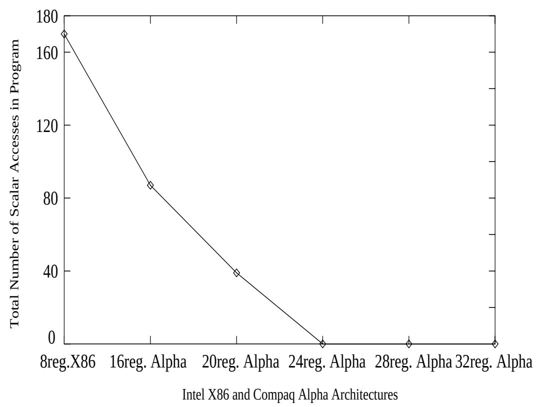

Figure 6: Scalar accesses for Alpha and Intel-X86

4.4 Scalar Data Remapping

We assume that the reorganization and separation of scalar and array accesses are compiler level tasks. As mentioned in Section 2, there are several cache reorganization options for scalars: vertical or horizontal partitioning [2], which partition the cache along cache ways and lines, respectively. Another option is to use a scratchpad SRAM area and direct the scalar memory accesses to this partition. Here, an appropriate partitioning option must be selected. Vertical partitioning schemes are wasteful: the existing cache has to be partitioned into a power of two, and our results indicate that the memory footprint of the scalars in embedded media applications is small. Instead, we advocate the use of a

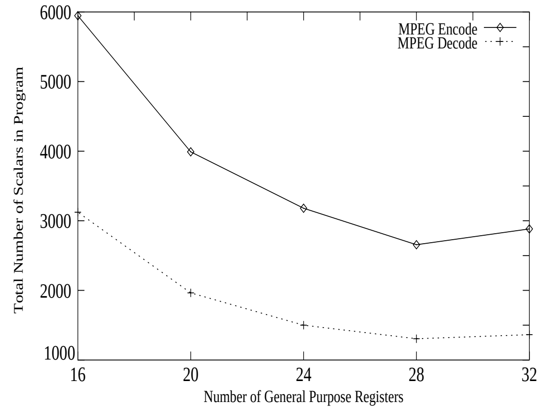

Figure 7: Effect of register coloring heuristic

scratchpad SRAM area. Separate SRAMs are widely used in DSP's: they are typically used to hold frequently used data such as floating point constants. A scratchpad SRAM guarantees single cycle access time to scalars since there are no cache misses. Moreover, the scratchpad on-chip SRAMs have small sizes, making this scheme ideal for data with small memory footprint such as the scalars in embedded media applications. This is also beneficial for a software-directed approach, since as shown in [18] every hardware partition can be logically partitioned and the scalar buffer area can be implemented as a logical partition. We assume the SRAM area to be sufficient to hold all the scalar data. No architectural modifications are necessary since many embedded processors have a scratchpad buffer area, see Table 1.

Therefore, if the embedded processor is equipped with a scratchpad SRAM area, the scalar memory accesses can be annotated by the compiler and remapped to the scratchpad. If not, then the Instruction Set Achitecture

	Baseline	Miss (%)	Partitioned	Improvement(%)	Scalars(%)	Warmup
EPIC	1753939	13.6	1589065	9.5	32.0	15598
G721 Encode	1377675	2.0	1369395	0.6	4.5	9
GSM	239914	0.5	230549	3.9	2.3	19
JPEG Encode	228644	9.3	224185	1.9	1.1	21
RASTA	216173	6.9	203373	5.9	16.0	59

Table 4: The number of misses for the baseline and with the scalar SRAM buffer are shown in the first and third columns, respectively. The second column shows the baseline miss rate. The percent drop in miss rate for remapping scalars to scratchpad is given in the fourth column. The fifth column is the percentage of scalar accesses to total memory accesses. The last column shows the scratchpad buffer warmup costs, i.e., the cost associated with promoting scalars from main memory to the scratchpad SRAM.

(ISA) can be augmented by special load-store instructions which would channel the scalar data to a separate cache area. The modifications to the compiler are minimal and consist of statically determining the application memory size and mapping the scalar accesses to the special load-store instructions.

We ran the Mediabench benchmarks with the baseline cache settings; we compared this with the same cache settings but with the scalar accesses being redirected to the SRAM buffer area by the compiler. We stress that capacity misses are not an issue here: Fritts et al. [10] have shown that data working set sizes of the considered benchmarks are very small. The results for selected benchmarks are presented in Table 4. The improvement depends on the particular benchmark and ranges between 0.6 to 9.5 percent. This improvement is more pronounced for the benchmarks which have a significant percentage of scalars in their memory accesses. Our results also affirm that scratchpad warmup costs are extremely small compared to the number of cache misses.

We also replicated our experiments for a 2 Kbyte 2-way cache organization. Table 5 shows the results. Note that the percentage improvements due to remapping of scalars to scratchpad are similar to the direct mapped cache results.

	Miss Rate(%)	Improvement(%)
EPIC	12.7	6.0
G721	1.2	0.1
GSM	0.5	5.5
JPEG	5.9	1.6
RASTA	5.3	9.2

Table 5: The results for the 2-way associative cache. The first column shows the baseline miss rate. The percentage reduction on miss rate due to remapping is shown in the second column.

5 Conclusion and Future Work

We have performed an analysis of scalars in embedded systems. We established the memory requirements of scalars in embedded applications and presented a compiler algorithm to extract this information. We then discussed several architectural issues pertaining to scalars in embedded systems.

This is ongoing work in line with our vision of creating memory systems with logical partitions where accesses are being mapped based on their static properties [18]. We are currently in the process of expanding our initial research. We will integrate our technique with other compiler/architectural techniques that handle diverse types of memory accesses.

References

[1] Appel A. W., George L., "Optimal Spilling for CISC Machines with Few Registers", *To appear in: ACM Sigplan Conference on Programming Language Design and Implementation*, June 2001

[2] Albonesi D. H., "Selective Cache Ways: On-Demand Cache Resource Allocation", *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999

[3] Belady L. A., "A Study of Replacement Algorithms for a Virtual-Storage Computer", *IBM Systems Journal*, 5(2), 1966

[4] Bhandarkar D. P., "Alpha Implementations and Architecture, Complete Reference Guide", pp.42-43, *Digital Press*, 1996

[5] B.Bishop B., Kelliher T., Irwin N., "A Detailed Analysis of MediaBench",*1999 IEEE Workshop on Signal Processing Systems*, Taipei, Taiwan, November 1999

[6] Burger D., Austin T. D., "The Simplescalar Tool Set, Version 2.0", *University of Wisconsin-Madison Computer-Sciences Department Technical Report #1342*, June 1997

[7] Burlin J., "Optimizing Stack Frame Layout for Embedded Systems", *Masters Thesis, Computing Science Department, Uppsala University*, Uppsala, Sweden, November 2000

[8] Cooper K. D., Harvey T. J., "Compiler-Controlled Memory", *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Systems (ASPLOS)* October, 1998

[9] Engblom J., "Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools", *ACM Sigplan Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999

[10] Fritts J., Wolf W., Liu B., "Understanding Multimedia Application Characteristics for Designing Programmable Media Processors", *Multimedia Hardware Architectures, Proceedings of SPIE*, San Jose, CA, January 1999

[11] Kulkarni C., Catthoor F., H. De Man, "Advanced Data Layout Organization for Multi-media Applications", *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM 2000)*, Cancun, Mexico, May 2000

[12] Lee C., Potkonjak M., Mangione-Smith W. H., "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 30th Annual International Sym-*

posium on Microarchitecture, December 1997

[13] Lee H. S., Tyson G. S., "Region-Based Caching: An Energy Delay Efficient Memory Architecture for Embedded Processors", *Proceedings of PACM (CASES'00)*, San Jose, California, November 2000

[14] http://www.eecs.harvard.edu/hube/software/software.html

[15] Memik G., Kandemir M., Haldar M., Choudhary A., "A Selective Hardware/Compiler Approach for Improving Cache Locality", *Northwestern University Technical Report CPDC-TR-9909-016*, 1999

[16] Milutinovich V., Tomasevic M., Markovic B., Tremblay M., " The Split Temporal / Spatial Cache: Initial Performance Analysis", *Proceedings SCIzzL-5,* Santa Clara, California, March 1996

[17] Moritz C. A. , Frank M., Amarasinghe S.,"FlexCache: A Framework for Compiler Generated Data Caching", *Second Workshop on Intelligent Memory Systems, IRAM00, Held in Conjunction with ASPLOS11*, Cambridge, Massachusetts, 2000

[18] Moritz C. A. , Frank M., Amarasinghe S.,"FlexCache: A Framework for Compiler Generated Data Caching", *To appear in Lecture Notes in Computer Science, Springer-Verlag*, 2001

[19] Mueller F., "Compiler Support for Software-Based Cache Partitioning", *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, California, June 1995

[20] O'Boyle M., Knijnenburg P. "Non-Singular Data Transformations: Definition, Validity, Applications", *Proceedings 6th Workshop on Compilers for Parallel Computers (CPC'96)*, Aachen, Germany, 1996

[21] Panda P. R., Dutt N. D., Nicolau A., "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications", *Proceedings of European Design and Test Conference*, Paris, France, 1997

[22] Ranganathan P., Adve S., Jouppi N. P., "Reconfigurable Caches and Their Application to Media Processing", *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, June 2000

[23] http://suif.stanford.edu/

Algorithm 1 Find Routine Scalar Memory Requirement from CFG

```
Require: localScalarSize of block
  /* Phase 1 */
  /* For each routine, traverse its control flow graph */
  for each routine do
    calculate scalar bound for each basic block;
    mark back edges in CFG;
    /* Add the entry back block to workList */
    E = entry basic block;
    E.scalarBound = E.localScalarSize;
    E.resolved = true;
    workList = successors of E;
    while !empty(workList) do
      B = next element in workList;
      allResolved = true;
      maxBound = 0;
      /* check if all B's predecessors are resolved */
      for each predecessor P of B do
        if the edge (P,B) is not marked and P is not resolved then
          allResolved = false;
          break;
        else
          maxBound = max(maxBound, P.scalarBound);
        end if
      end for
      if allResolved then
        remove B from workList;
        B.resolved = true;
        B.scalarBound = maxBound + B.localScalarSize;
        add all unresolved successors of B to workList;
      end if
    end while
    set localScalarSize of the current routine as scalarBound of its exit block.
  end for
```

Algorithm 2 Find Program Scalar Memory Requirement From Call Graph

```
Require: localScalarSize of routine
  /* phase 2 */
  /* Traverse call graph*/
  mark back edges in call graph;
  M = main routine;
  M.resolved = true;
  M.scalarBound = R.localScalarSize;
  M.resolved = true;
  workList = successors of M;
  while !empty(workList) do
    R = next element in workList;
    allResolved = true;
    maxBound = 0;
    /* check if all R's predecessors are resolved */
    for each predecessor P of R do
      if the edge (P,R) is not marked and P is not resolved then
        allResolved = false;
        break;
      else
        maxBound = max(maxBound, P.scalarBound);
      end if
    end for
    if allResolved then
      remove R from workList;
      R.resolved = true;
      R.scalarBound = maxBound + R.localScalarSize;
      add all unresolved successors of R to workList;
    end if
  end while
  The program scalar buffer bound is the maximum scalarBound of all routines;
```