Presented at the SRC TECHCON 2003 Symposium

Memory Profiling for 3G Domain DSP Development

Hillery C. Hunter Chien-Wei Li Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{hhunter, cwli, hwu}@crhc.uiuc.edu

Abstract

This paper describes and demonstrates a set of memory profiling tools implemented in the IMPACT compiler framework. These tools track producer/consumer relationships, heavily utilized arrays, stream and strided accesses, unused memory space, and other metrics. We show that a memory profile can provide profitable feedback to both application developers and architecture/compiler researchers at multiple stages of development, including rapid proof-of-concept and validation for new architecture and compiler techniques.

1. Introduction

Implementation of Third Generation (3G) telecommunication/media systems has greatly increased the complexity of programming for handheld devices. Hand assembly coding has become prohibitively expensive due to application size, increased processor issue width, and demands for short turnaround times. As a result, high-level language programming and compilers are becoming essential to successful use of DSPs in this domain.

While application development in high-level languages (HLLs; primarily C) greatly speeds time-to-market, it constitutes a fundamental shift in programmer mind-set. Whereas an assembly programmer aims to optimally map algorithms to processor resources, a C programmer must express kernels as functions, pointers, and data structures, often with little knowledge of how the compiler will schedule operations and allocate data on the target platform. Inefficient HLL data usage and generic procedure invocations may make it difficult, or even impossible, for a compiler to automatically detect optimization opportunities and schedule for special features like circular buffers and non-standard memory hierarchies. Some compilers implement these optimizations with programmer directives; however, if HLL source is obtained from a vendor, addition of directives is often difficult because the vendor's code structure and algorithms are unfamiliar. Existing profiling tools [1, 2, 3] have been designed primarily with performance tuning and hand code optimization in mind; we have found detailed memory profiling to be useful to compiler development and compiler/architecture co-design.

The following section briefly describes the memory profiling tools implemented in the IMPACT [4] framework. Insights are then discussed in the areas of application characterization, buffer utilization, memory partitioning, and compiler development for a novel embedded architecture. We conclude with a discussion of related work.

2. Profiling Environment

The described profiling mechanisms were implemented in the IMPACT infrastructure. IMPACT is an optimizing *instruction-level parallelism* (ILP) compilation framework which includes tools for emulation, performance profiling, simulation, and now, detailed dynamic memory analysis. In this paper, insights are presented from the telecommunication applications listed in Table 1; video and image algorithms such as JPEG, JPEG-2000, MPEG4, and H.263 have also been studied, but are omitted due to space constraints.

In the implemented tools, program memory is viewed as *objects*; these may be stack spaces, heap objects, or global variables. For each object, a structure is maintained that records its starting address and the amount of memory it has been allocated. Statistics on object usage and access are preserved while a memory section is *live*, or available for program usage. As code is emulated and run on the host platform, a trace of `load` and `store` accesses to memory objects is recorded in the form of summary *runs*. Runs group together accesses with a common property, *e.g.* same-size access to sequential memory addresses or accesses occurring with a uniform stride. Default profiling includes all memory traffic, but gathered information may be restricted to particular loops, individual source code operations, or specified data structures so as to improve speed.

Various analyses may be conducted as the memory profile is gathered, or by post-processing the profile information. Profile data can also be annotated onto the compiler's intermediate representation to facilitate optimizations and transformations performed at later compilation stages. Possible code annotations include marking loops with producer-consumer relationships and flagging source-code arrays which account for high volumes of memory traffic.

3. Utility of Memory Profiling for Compiler/Architecture Co-Design

A memory profile can prove useful in several ways. Aside from benefits for hand optimization of code (like those in [2]), we discuss insights here for architecture and compiler research and design.

Table 1: Telecommunication applications studied.

Applications	Source	Description
adpcmdec *adpcmenc*	MediaBench [6]	Intel/DVI ADPCM codec
g721dec *g721enc*	MediaBench	Voice compression according to the CCITT G.721 standard
g724dec *g724enc*	ETSI	GSM 06.60 EFR speech transcoding [5] (digital cellular communication)
gsmdec *gsmenc*	MediaBench	Lossy sound compression according to the GSM 6.10 RPE-LTE standard [7]

Characterizations

Profile information can be used to create detailed application characterizations which may lead to both hand optimization opportunities and domain-specific compiler and (micro-)architectural innovations. As a simple example, if an application is found to have regular memory access patterns, stride and contiguous access information can be used as impetus for specialized compiler-directed prefetching. Characterizations developed as part of this work include:

- stride distances;
- contiguous run lengths;
- producer-consumer relationships; and
- memory traffic bursts.

Among other insights, examination of memory usage characteristics has led to detection of both unused data and instruction memory space in reference applications. An example of unused data memory is shown in Figure 1. In this C code excerpt from the ETSI GSM 06.60 EFR decoder implementation [5], the `tmp` array is allocated 80 two-byte elements. However, the last 60 bytes of this array are never used, so this is wasted space which should not be allocated in the `Syn_filt` function stack.

Causes of unused instruction space were described in [8] for reference applications. While superfluous functions will not generally be present in custom code, when using 3rd party or legacy code, it is possible that functions could be present in the source which are not necessary for execution in the target environment. It can be assumed that a good compiler would remove functions which have no references, but an instruction memory/execution profiling tool could point the developer to referenced, but untouched, code which might not be needed for correctness in the target environment.

We have also used our tools to characterize memory *bursts*, or significant volumes of activity that occur within a given interval of cycles. Looking at total application burst behavior gives an indication of bus bandwidth and memory port utilization, but it has also proven useful to examine traffic due to individual objects, or variables, within an application. Figure 2 shows an excerpt of memory traffic to/from the gsmdec `LARp` array, which accounts for 20% of the application's total memory traffic. Access to this array of eight 2-Byte quantities occurs periodically, with approximately 10,000 cycle gaps between `load` intervals and 38,000 cycles between `store` intervals. This information could be used to direct compiler-assisted prefetching, but also points toward data layout optimization opportunities like those discussed in [9].

A more complete characterization can be seen in Figure 3, which shows memory burst traffic of g724dec arrays, categorized according to Table 2. Periodic access patterns are seen again, pointing toward memory partitioning such as that proposed by Benini for power saving or by Farrahi for memory sleep modes [9]. Farrahi *et al.* appear not to have had a framework for whole-application memory profiling, so our memory profiles demonstrate the feasibility of their ideas. As mentioned in [9], memory profiles are quite accurate for telecommunication applications. If used for optimizations which do not violate correctness (*e.g.* the processor will simply stall if sleep modes are miscalculated and a request must wait for a partition to awake from sleep mode), profiles can enable compiler techniques which would otherwise require expensive static analysis.

Rapid proof-of-concept

A second motivation for memory profiling is its ability to provide rapid proof-of-concept for new architecture and compiler ideas. While an accurate architecture simulator or complete compiler algorithm must still be developed, the profile provides quick first-level validation of proposed concepts.

As an example, array usage profiles have been developed to guide compiler development for our testbed clustered EPIC architecture [10]. Code generation for the proposed architecture requires new methods of program partitioning and unique mapping of code, data, and communication. Standard paths of performance evaluation thus require either extensive compiler development or hand coding. For this compiler/architecture co-design project, however, memory profile information provided a relatively early indication that the architecture was on the right track.

A primary goal of the testbed architecture is to map various types of memory traffic (*e.g.* look-ups, inter-procedural communication, and function state) to inexpensive on-chip communication paths, so as to avoid access to memory. Figure 4 shows categorized accesses for six applications. These results show that significant amounts of application traffic correspond to target access types, and thus demonstrate the potential use of the architecture's on-chip communication mechanisms.

Compiler validation

If a memory profile is used as proof-of-concept for an idea, the next step will be to implement necessary architecture mechanisms and compiler algorithms or heuristics. The original profile information can serve as a benchmark for judging the efficacy of implemented compiler techniques.

For example, in [11], we explored opportunities present in telecommunication reference codes for automatically reducing memory traffic by means of a small on-chip buffer similar to a compiler-managed scratchpad. Code for DSP chips with software managed scratchpad space, FIFO or circular buffers, or other specialized architectural structures, must often be generated by hand. For our study, small, commonly ac-

```
Syn_filt(coeffs[11],input[40],output[40],
            mem[],update) {
    short i, j, tmp[80], *yy;
    int s;

    yy = tmp;

    for (i = 0; i < 10; i++) {
       *yy++ = mem[i]; }

    for (i = 0; i < 40; i++) {
       s = input[i] * coeffs[0] * 2;
       for (j = 1; j <= 10; j++) {
          s = yy[-j] - ( s * coeffs[j] * 2 );}
       s = s << 3;
       *yy++ = round (s); }

    for (i = 0; i < 40; i++) {
       output[i] = tmp[i + 10]; }

    if (update) {
       for (i = 0; i < 10; i++) {
          mem[i] = output[30 + i]; }
    }
}
```

Figure 1: Simplified C code for orignal Syn_filt function from g724dec: `tmp` is allocated 30 extra elements.

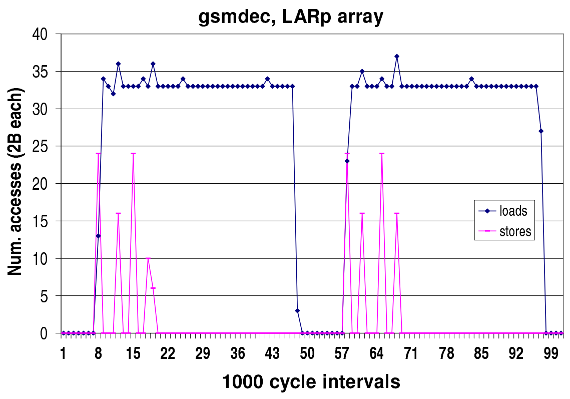

Figure 2: gsmdec LARp array memory traffic.

Table 2: Data object categories.

Category	Meaning
LT	Look-up tables (math short-cuts, constants, non-adaptive coefficients)
IO	Input/output
IP	Inter-procedural communication (passing intermediate values among functions)
ST	State (preserved between function invocations)
CO	Coefficients (adaptive)
TP	Temporary values within a single function (intra-procedural data, but too large to fit in register file)
OTHER	Access to non-array/structure data

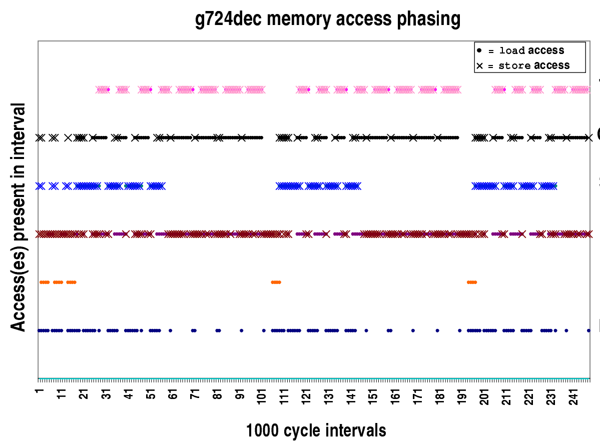

Figure 3: g724dec memory traffic phasing: arrays categorized according to Table 2.

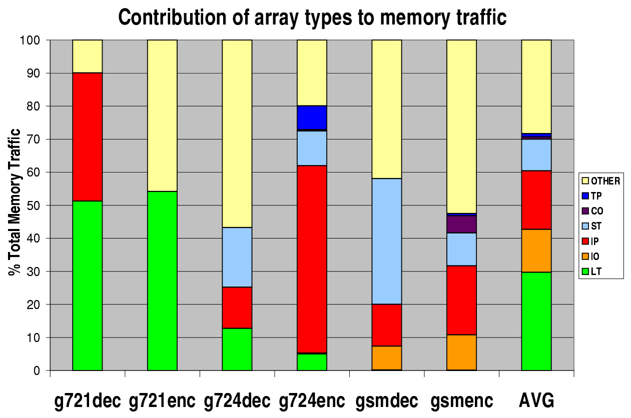

Figure 4: Memory traffic accounting by data object type (Table 2).

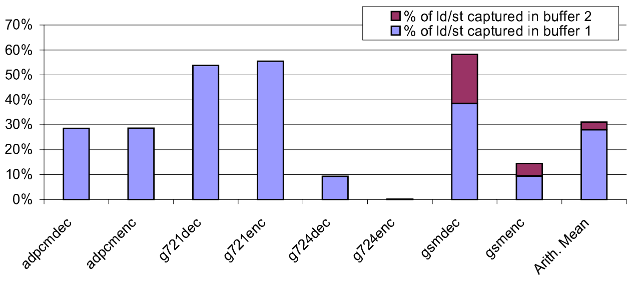

Figure 5: Percentage of memory traffic captured by array allocation to an on-chip buffer.

cessed arrays (filter coefficients and look-up tables) were first identified using a dynamic profile. Having used our memory profiling tools to demonstrate the potential for reduction in memory traffic (shown in Figure 5), we then developed compiler heuristics and algorithms to automatically select these critical arrays. In this process, we used the original profile information to validate that compiler heuristics were working correctly.

4. Related Work

Several commercial compilers for DSP/embedded processors already incorporate profiling, but these are primarily execution profilers intended for performance tuning: the Blackfin DSP [12] compiler package includes a function-level execution profiler; and the Green Hills ST100 DSP family CodeBalance tool [3] allows developers to enter constraints for tuning the performance vs. code size tradeoff of its variable-width instruction set architecture. Texas Instruments has released software for monitoring code coverage [1] and cache performance [2], but (in their commercial form) these tools appear to be restricted to actual TI product memory configurations. They are thus code optimization tools not intended to provide the user with architecture or compiler insight.

Within academia, the DTSE (data transfer and storage exploration) methodology described in [13] includes the four-part "Atomium" toolset. Atomium can be used to perform source-to-source C code transformations aimed at simplifying custom multimedia processor design. While there is some overlap between information provided by Atomium and IMPACT memory profiling, Atomium is a CAD-oriented tool for custom memory systems, and requires user input to specify transformations, memory hierarchy properties, and timing constraints.

Memory characterization and analysis studies have been published for several classes of 3G-type applications on general-purpose platforms. These studies, however, have a different focus than ours, since they concentrate on cache behavior, rather than use of profile information to improve the compilation process or highlight potential use of specialized architectural features. Previous characterization studies include theses by Fritts [14] and Slingerland [15].

[9] describes several approaches toward memory partitioning. These include specialized memory hierarchies, partitioning such that frequently accessed variables appear in a small memory, and grouping variables which are accessed temporally close to one another so as to enable sleep modes when a partition's variables are not needed. Some previous work assumes that data access patterns are known and only explores the partitioning problem; other work evaluates approaches using kernels. Our tool's ability to provide detailed memory profile information for applications allows exploration of memory performance and power techniques using high-level language code and whole applications.

5. Conclusion

We have found a detailed memory profile to be useful at multiple stages of compiler and architecture research. While memory profiling has clear utility as an aid for hand optimization of performance, instruction coverage, and data usage, it can also provide insights about application properties, aid first-level evaluation of new architecture and compiler ideas, and validate completeness of compiler algorithms and heuristics.

Acknowledgments

This work was supported by the Semiconductor Research Corporation under the grant "Memory Efficient EPIC/VLIW Architecture." Hillery Hunter was supported by a fellowship from the National Science Foundation. The authors would like to thank IMPACT contributors for their work on the compiler; particularly John Sias for his work on the initial memory profiler.

References

[1] Texas Instruments Incorporated, "Code coverage and multi-event profiler user's guide." Literature Number SPRU624A, Jan. 2003.

[2] Texas Instruments Incorporated, "Using cache analysis tool to improve cache utilization." Application Report SPRA863, Jan. 2003.

[3] Green Hills Software Inc., "Embedded ST100 development guide: MULTI 2000." Pub ID D16B-I0201-89NG, 2000.

[4] W. Hwu, R. Hank, D. Gallagher, S. Mahlke, D. Lavery, G. Haab, J. Gyllenhaal, and D. August, "Compiler technology for future microprocessors," *Proc. IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.

[5] ETSI TC-SMG, "Digital cellular communications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)," Tech. Rep. ETS 300 726, European Telecomm. Standards Institute, Mar. 1997.

[6] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 330–335, Dec. 1997.

[7] ETSI TC-SMG, "GSM full rate speech transcoding (GSM 06.10)," Tech. Rep. Version 3.2.0, Release 92, Phase 1, European Telecommunications Standards Institute, Feb. 1992.

[8] H. Hunter and W. Hwu, "Code coverage and input variability: effects on architecture and compiler research," in *Proceedings of the Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 79–87, Oct. 2002.

[9] L. Benini, A. Macii, and M. Poncino, "Energy-aware design of embedded memories: a survey of technologies, architectures, and optimization techniques," *ACM Transactions on Embedded Sytems (TECS)*, vol. 2, pp. 5–32, Feb. 2003.

[10] J. Sias, H. Hunter, W. Hwu, and N. Carter, "Rationale for an EPIC media and signal processing architecture," Tech. Rep. IMPACT-00-01, University of Illinois, 2000.

[11] H. Hunter, C.-W. Li, N. Carter, and W. Hwu, "Capturing telecommunication application memory traffic," Tech. Rep. IMPACT-03-01, University of Illinois, 2003.

[12] Analog Devices, Inc., Digital Signal Processor Division, "VisualDSP++ 3.0 C/C++ compiler and library manual for Blackfin^{TM} DSPs." Publication 82-000410-03, Apr. 2002.

[13] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom memory management methodology*. Boston, MA: Kluwer Academic Publishers, 1998.

[14] J. Fritts, *Architecture and compiler design issues in programmable media processors*. PhD thesis, Princeton University, 2000.

[15] N. Slingerland, "Architectures for multimedia," Master's thesis, University of California at Berkeley, 2000.