Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example

Ann Gordon-Ross, Susan Cotterell and Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
http://www.cs.ucr.edu/~vahid
*Also with the Center for Embedded Computer Systems at UC Irvine

Abstract--**Embedded systems commonly execute one program for their lifetime. Designing embedded system architectures with configurable components, such that those components can be tuned to that one program based on a program pre-analysis, can yield significant power and performance benefits. We illustrate such benefits by designing a loop cache specifically with tuning in mind. Our results show a 70% reduction in instruction memory access, for MIPS and 8051 processors – representing twice the reduction from a regular loop cache, translating to good power savings.**

Keywords--**Loop cache, architecture tuning, low power, fixed program, embedded systems.**

I. Introduction

Microprocessor usage in many embedded systems has an important feature distinct from usage in "desktop" computing systems like personal computers, laptop computers, and servers. In desktop systems, the microprocessor executes many different programs over the system's lifetime. In contrast, in many embedded systems such as digital cameras or automobile control systems, the microprocessor may execute one program for the system's lifetime – the program is *fixed*.

Microprocessor architects have long been aware of this fixed program feature. They have thus incorporated certain configurable features into the microprocessor architectures. For example, a microprocessor chip may support two configurations: one using an external program memory, another using a smaller on-chip memory, which frees the external pins for parallel I/O use. The embedded system developer configures the chip during system initialization, using the second configuration only if the fixed program fits in on-chip memory. Architects also identify classes of programs, and develop separate architectures for each class – perhaps differing in the amount of on-chip memory, on-chip peripherals, supported interrupts, etc.

Continued increases in chip capacity have enabled far more extensive incorporation of configurable features into a microprocessor architecture. One reason is because developers today often acquire a microprocessor architecture in the form of intellectual property, known as a core, and they then integrate the microprocessor into a system-on-a-chip. Thus, they can tune the core's configurable components to match the fixed program, before fabricating the chip. For example, they can create just the right amount of on-chip cache or choose between direct-mapped or set-associative cache. A second reason is that the cost of including additional transistors onto pre-fabricated microprocessor chips has decreased – an important point in the cost hypersensitive embedded systems market. Transistors are by no means free, but the cost equations have changed such that including some additional transistors may not impact the chip cost as much as it did in the past. Thus, we are seeing an increase in on-chip configurable components, such as configurable on-chip cache [9].

Microprocessor architects can therefore increasingly consider creating aggressively parameterized components for allowing an embedded system developer to tune those components to a fixed program. We will demonstrate the effectiveness of such aggressive parameterization using a loop cache. We will show one way to redesign a basic loop cache to exploit the fixed program feature. We provide results showing excellent reductions in instruction memory access, translating to reduced power.

II. Architecture Tuning

We first describe the basic steps that a developer might follow in tuning an architecture to a fixed program. Such tuning is the complement of the common task of tuning a program to an architecture – something that has long been done in both embedded and desktop systems.

Architecture tuning is a step added to the end of the program development process. *Architecture tuning* is the task of selecting the best configuration for a particular fixed

Manuscript submitted: 21 Dec. 2001.
Manuscript accepted: 20 Jan. 2002.
Manuscript received: 28 Jan. 2002.

program, where best is a user-defined combination of power, performance, size, and other metrics. We define a *configuration* as a particular setting of all parameters in all parameterized components in an architecture. Architecture tuning can be done pre-fabrication (in the case of cores) or post-fabrication. Architecture tuning consists of several sub-tasks [12]:

- *Profiling* consists of characterizing the program as it executes with representative input vectors.
- *Evaluation* consists of determining the power and performance (and size for pre-fabrication tuning) metrics for a given program and configuration. Evaluation can be done by in-system execution and measurement, or by simulation-based methods.
- *Exploration* consists of strategies to efficiently explore the potentially enormous set of possible configurations in an effort to find the set of configurations representing interesting power/performance tradeoffs. Iteration among the evaluation and exploration phases is likely.

A very aggressive form of architecture tuning involves creating customized instructions, either pre-fabrication [3][4] or post-fabrication [6]. Our focus is on tuning that does not modify the instruction set.

After architecture tuning, a developer will likely need to modify the original program boot sequence to include configuration of the architectural parameters with the selected configuration. In a post-fabrication approach, the developer then downloads the new program into existing chips. In a pre-fabrication approach, the developer creates a customized version of the microprocessor and proceeds with chip fabrication steps. The developer may choose to iterate between architecture tuning and program tuning, as they are interdependent.

III. A Loop Cache Example

An embedded microprocessor architect, aware that the eventual developer will perform architecture tuning, can design tunable components to provide power and performance benefits, as we will now demonstrate.

A. Dynamically Loaded Loop Cache

We use a loop cache as an example. We'll focus in this paper on a loop cache's ability to reduce power, but improving performance could also be considered. Fetches to the first level of instruction memory (whether cache or regular memory, whether on-chip or off-chip) typically account for much of a system's power consumption [4], because such fetches occur frequently, require driving high-capacitance bus lines (especially when off-chip), and may involve numerous tag comparisons.

At the same time, many embedded system programs spend much of their time in small, tight loops [7][13]. Thus, numerous approaches to reducing instruction fetch power focus on caching such small loops in a very small and hence very low-power *loop cache*, having perhaps only 16-64 entries. Power per access may be 50-100 times less than access to regular instruction memory [8].

The problem remains as to how and when to fill and fetch from a loop cache. Some architectures have special instructions for forming loops [1][11], which can be used to trigger loop cache fills and to know when the loop terminates, but such approaches are not applicable to existing architectures without such instructions, and also limit loop forms. Kin [5] proposed a very small regular cache called a filter cache, having tag comparison and miss logic, placed before instruction memory. To reduce the many misses of a filter cache, Bellas [2] proposed a modified architecture coupled with program tuning, wherein a profile-guided compiler would map only the most frequent instructions to a special memory region recognized by the architecture as destined for the filter cache; all other instruction accesses would bypass the filter cache. The result was fewer misses and thus less performance overhead and greater power savings.

Lee [7][8] introduced a loop cache with no performance overhead and requiring no profile-guided compilation. A loop cache controller fills the cache after detecting a simple loop – defined as any short backwards branch instruction. The fill occurs by copying the dynamic instruction stream – no stall is generated. Once the fill is a complete, the loop cache controller seamlessly switches to loop cache fetching, shutting down all logic associated with regular instruction memory fetching. The controller conservatively aborts the fill or exits loop cache fetching if a jump is taken within the loop, since a taken jump means we may not fill the entire loop or we may actually be leaving the loop. Note that predicated instructions increase the number of loops without internal jumps. The loop cache controller uses a simple wrap-around counter for indexing into the cache, and requires no tags. Average instruction memory access reductions were 40% using a 32-entry loop cache with an M*CORE processor (larger sizes did not yield further improvements since most loops were small), with overall system power reductions of 14% [8].

B. A Pre-Loaded "Tunable" Loop Cache

1) Overview

We use the Lee-style loop cache as our starting point. While that loop cache reduces average instruction access power in many examples, it could reduce power further in several examples. Some limitations reduce a dynamically loaded loop cache's power savings:

- Many loops contain internal branches – especially in architectures without predicated instructions.
- Some loops are not formed by a single backwards branch – they may consist of several backwards branches pointing to the loop start.

- Nested loops could cause loop cache thrashing.
- Some programs have subroutines that contribute to much execution time.

If we know architecture tuning to a fixed program will occur, we can design a loop cache that largely overcomes these limitations. We design a loop cache that gets *pre-loaded* with the best loops as determined by profiling. Thus, during program execution, the loop cache contents will not change. Pre-loading the loop cache comprises the architecture-tuning step of development. Pre-loading can be carried out by several alternative means, one of which is to create a memory-mapped register that can be used to push items into the loop cache, and then filling the loop cache as part of program initialization.

Pre-loading has several advantages. First, loops with branches can be pre-loaded, since the fill is occurring offline and not during dynamic program execution – we simply load all instructions within an address range. Second, loops constructed with multiple backwards branches can be detected through pre-analysis and included in the cache. Third, only the best loops can be pre-loaded, including the best level of a nested loop sequence. Fourth, subroutines can be included. Like a Lee-style loop cache, a pre-loaded loop cache requires no special compiler or binary modification tool.

A pre-loaded loop cache has a disadvantage of limiting the number of loops that can be cached. In some cases, two or three small loops contribute to the majority of program execution, so this disadvantage sometimes is not an issue [13]. A second disadvantage is that a simple counter-based indexing approach won't work due to the branches. We solve this by computing the index for a loop *L* as (*PC – Lma)* + *Lca*, where *Lma* is the loop's start address in memory, and *Lca* is the loop's start address in the loop cache. Although we pre-compute *Lma – Lca*, this subtraction-based approach consumes a bit more power than the counter approach. A third disadvantage is the need for the tuning step.

2) Pre-Loaded Loop Cache Control

A key difference between a dynamically loaded loop cache and pre-loaded one lies in how the loop cache controller decides to enter and exit the loop cache fetching state.

As for entering, in a pre-loaded loop cache, rather than switching to loop cache operation upon detecting a short-backwards branch, the controller instead, upon execution of any taken jump instruction, switches if the next address falls within the starting and ending addresses of one of the loops in the cache. Thus, the program initialization sequence must write the starting and ending addresses of the loops in the cache into a set of registers in the controller, which are connected to comparators. Such comparison does consume additional power, but that power is moderated by two factors. First, there are only a few such registers, perhaps with only three or four. Second, the comparisons only occur when we are not fetching from the loop cache, which ideally is only a small percentage of time.

TABLE 1
BENCHMARK DESCRIPTIONS

Benchmark	Size of assembly in bytes		Description
	MIPS	8051	
adpcm	7648	na	Voice Encoding
bcnt	3884	5430	Bit Manipulation
binary	3696	403	Binary Insertion
blit	4180	5969	Graphics Application
brev	3976	2497	Shifting and Or Operations
compress	7480	na	Data Compression Program
crc	4248	831	Cyclic Redundancy Check
des	6124	na	Data Encryption Standard
engine	4440	na	Engine Controller
fir	4232	na	FIR Filtering
g3fax	4384	8308	Group Three Fax Decode
jpeg	5968	na	JPEG Compression
matmul	3796	843	Matrix Multiplication
summin	4144	1648	Handwriting Recognition
ucbqsort	4848	3066	U.C.B Quick Sort
v42	6396	na	Modem Encoding/Decoding

As for exiting, the controller must quickly determine whether a branch in a loop causes a loop exit. We use pre-analysis to compute the branch target when possible. We associate two extra bits with each instruction in the loop cache. One bit configuration indicates that the instruction can never exit the loop. A second bit configuration indicates that the instruction is a jump that exits the loop if the jump is taken. A third configuration indicates that the instruction is a jump that exits the loop when not taken (i.e., the end of the loop). A fourth configuration indicates that loop cache operation should always terminate after this instruction. This fourth option is present because sometimes we cannot determine the target address of a branch, such as in the case of an indirect jump. In those cases, we conservatively exit loop cache operation. Interrupts also cause the controller to terminate loop cache operation for the current loop until the next loop is detected – a loop that could even be part of the interrupt service routine itself.

IV. EXPERIMENTS

We evaluated examples to determine the advantages of using a pre-loaded loop cache over a dynamically loaded one. We evaluated instruction access reductions for two popular embedded microprocessors, a 32-bit MIPS and an 8-bit 8051 processor. We modified MIPS and 8051 instruction-set simulators to generate trace files, and developed loop-cache simulators to process those traces and record the activity related to loop cache filling and fetching, the loop cache controller operations, and the instruction memory accesses. We used examples from the PowerStone benchmarks [9], with TABLE 1 providing their names, size in bytes, and a short description of each. The figure shows that some of the

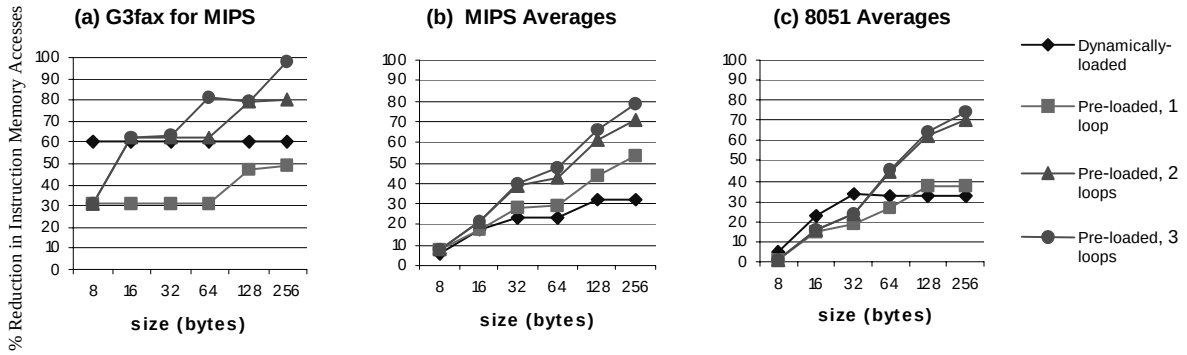

Fig. 1. Instruction memory access reductions for (a) one benchmark on the MIPS, (b) MIPS averages, (c) 8051 averages

benchmarks were not run on the 8051, mostly due to the data memory limitations of the 8051.

Fig. 1(a) shows results for one benchmark, for a dynamically loaded loop cache versus pre-loaded caches supporting one, two or three loops, for cache sizes ranging from 8 to 256 entries. Fig. 1(b) shows the average results for all the benchmarks on the MIPS, and Fig. 1(c) the average results for the 8051. On average, the dynamically loaded loop cache shows a 30-35% reduction in instruction memory accesses, slightly less than the 40% in [8] due to the lack of predicated instructions in the MIPS and 8051. We observed the same leveling off of improvement beyond size 32 as observed in [8]. The pre-loaded loop cache, in contrast, reduces instruction memory access by 70-80%, for both the MIPS and 8051 processors. These reductions come mostly from support of loops with internal branches and the inclusion of subroutines.

We evaluated the impact of such reductions on instruction fetch power savings, considering instruction memory access, loop cache fills and fetches, and loop cache controller operation, from a combination of gate-level power measurements using Synopsys synthesis and evaluation tools [10] and processor power evaluators based on instruction-level simulators. We ranged the power ratio of internal net switching to bus/memory activity from 50:1 to 400:1 to account for different technologies. The instruction fetch power savings for a dynamically loaded cache ranged from 20% to 30% for all cache sizes and ratios. The savings for a pre-loaded cache ranged from 30-35% for size 32 to 50-65% for size 256. As instruction fetch power often accounts for a large percentage of total power (e.g., 50% in [8]), reductions in instruction fetch power can yield good overall power reductions.

By performing analysis similar to above, a microprocessor architect may decide on a particular size and number of loops for a pre-loaded loop cache, possibly creating a multi-segment loop cache store where segments could be deactivated. An embedded system developer could choose among a dynamically loaded and pre-loaded loop cache depending on the fixed program, and could configure the cache to use the most appropriate size and number of loops.

V. Conclusions

Our results illustrate the benefits of microprocessor architect designers exploiting the fixed program feature of embedded systems, and of developers adding an architecture-tuning task to the development process. Extensive future work remains in this area of architecture tuning based on program pre-analysis. We are currently investigating results for larger benchmarks (e.g., MediaBench), and extensions to efficiently support more loops, including a hybrid dynamic/pre-loaded approach, a two-level pre-loaded loop cache, and efficient loop address detections using multi-stage comparisons.

Acknowledgments

This work was supported by the National Science Foundation (grant #CCR-9876006).

References

[1] ADSP-2106x User's Manual, Analog Devices, 1998.
[2] Bellas, N., Hajj, I., Polychronopoulos, C., Stamoulis, G. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. Int. Conf. on Computer Design, pp. 378-383, 1999.
[3] Fisher, J. A., Customized Instruction Sets for Embedded Processors. DAC, pp 253-257, 1999
[4] Gonzalez, R.E. Xtensa: A Configurable and Extensible Processor. IEEE Micro, pp. 60-70, 2000.
[5] Kin, J., M. Gupta, W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. Int. Symp. on Microarchitecture, pp. 184-193, Dec. 1997.
[6] Kucukcakar, K. An ASIP Design Methodology for Embedded Systems. Int. Workshop on Hardware/Software Codesign, pp. 17-21, 1999.
[7] Lee, L. H., W. Moyer, J. Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. Int. Symp. on Low Power Electronics and Design (ISLPED), August 1999.
[8] Lee, L. H., W. Moyer, J. Arends. Low-Cost Embedded Program Loop Caching – Revisited. Univ. of Michigan Technical Report CSE-TR-411-99, December 1999.
[9] Malik, A., B. Moyer B. and D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. International Symposium on Low Power Electronics and Design. June 2000.
[10] Synopsys, http://www.synopsys.com.
[11] TMS320C2x User's Guide, Texas Instruments, 1993.
[12] Vahid, F., T. Givargis. Platform Tuning for Embedded System Design. IEEE Computer, Vol. 34, No. 3, pp. 112-114, March 2001.
[13] Villarreal, J., R. Lysecky, S. Cotterell, and F. Vahid. Loop Analysis of Embedded Applications. UC Riverside Tech Report UCR-CSE-01-03.