Efficient and Effective Simulation of Memory Maps for System-on-Chip

Gabriele Luculli

Abstract

The design of complex system-on-chip (SOC) requires new methods and tools for the optimization of embedded software which is executed on ever more complex hardware architectures. The tuning of the memory subsystem is particularly difficult due to the many design parameters which are involved and the long time which is required to simulate different design configurations. In this paper, we propose a very effective mechanism for the simulation of generic memory maps on architectures with instruction and/or data cache memory. An important characteristic of our implementation is its large flexibility: any memory map and any cache configuration can be simulated without the need to modify or to re-compile the application code. We implemented such mechanism in our ISA retargetable environment and we showed that it loosely impacts the simulation performance.

1 Introduction

In the past, designers were not used to include cache memories in embedded products due to the difficulties of having a predictable behaviour for this kind of memory. Recently, the situation has started to change. New embedded products are often microprocessor-centric designs, where one or more microprocessors communicate by complex interconnects and share several main memories. The past slow performance cpu have left the place to high-performance, several hundred million hertz cores more and more hungry of memory bandwidth. As it already happened in general-purpose computing, cache memories have been proposed as the ideal solution to reduce the cpu-memory latency gap.

The adoption of cache memories in embedded products is a valuable solution from the hardware point of view, but it may be a real nightmare for the engineers who have to optimize the software. Cache behaviour is hardly predictable, and it is very difficult to establish a relation between a generic modification of the software code and the new cache performance. Moreover, quite a lot of time is required for any additional simulation. First, the software code has to be updated, compiled and linked. Then, it can be simulated by an instruction-set simulator which produces the cache statistics and, therefore, a quantitative evaluation of the software improvement.

Several code placement techniques have been proposed as possible ways of manipulating automatically the source code. Automation is useful, of course, to master the complexity of code and data optimization and to explore non-intuitive solutions. However, it doesn't necessarily improve the time required to derive a better layout.

In this paper, we propose a mechanism which enables the simulation of many different memory layouts (i.e. a memory layout is a mapping of the code or data address space into the memory address space) without the need of any physical modification of the source or binary code. In this way, a large set of evaluations can be safely done in short time and without compiling, linking or tracing a modified version of the application. The proposed mechanism is flexible enough to apply to both code and data caches.

In section 2 we briefly review the literature on mechanisms and methods used to evaluate different code placements. In section 3 we describe our translator of memory maps and its implementation in our RTOOLS retargetable environment [6]. Some experiments are reported in section 4 and section 5 concludes the paper.

2 Related Work

Many reordering algorithms have been proposed in order to obtain program layouts which better utilize the available memory hierarchy. They differ based on: 1) the *granularity* assumed by the algorithm, and 2) the *conflict model* used to guide the memory placement. Some of them apply to code [7, 10, 4, 8], while others to data [2, 11, 3]. In the past, the improvement of cache memory performance was the main issue [7, 10, 4, 2], today we start to see that other kind of memories, such as DRAM, may have a significant role in the optimization of system performance as well [3].

The effects of several kinds of granularity have been explored. For code placement, procedures and basic blocks

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

are by far the most natural code boundaries which have been considered [7, 4]; however, some results refer to more complex code structures such as basic block traces [10] or global control-flow graphs where the procedure boundaries are disregarded [8]. Global variables, arrays and local variables allocated in the program stack are instead the typical objects considered by data placement algorithms [2, 11, 3]. Different granularities usually refer to different degrees of code or data locality so, depending on the specific program characteristics, all may contribute to improve memory performance.

The main expected effect of any placement algorithm for cache memory is to improve the miss rate by reducing the experienced conflict misses. While the evaluation of the miss rate can be easily done by simulation techniques, the direct evaluation of the number of conflict misses is much more difficult to achieve. For this reason, practically all the proposed placement algorithms rely on some sort of model to estimate the possible conflict effects in the cache and, then, to guide the memory placement. Moreover, most of the times such model is a graph which has been annotated with some kind of information on the edges or the nodes.

Pettis et al. [9] adopt a procedure call graph which has been annotated with the number of calls that occurred between pairs of procedures. The analysis of one or more *execution traces* is used to derive such figures. A quite similar approach is adopted by Gloy et al. [4] to represent temporal ordering information among procedures. In [10] a weighted control-flow graph is exploited to select the most frequent sequences of basic blocks; in [3] an access graph is used to grasp the locality properties of a set of variables and, in a quite similar way, it is done by a temporal relationship graph in [2]; finally, a call graph which has been annotated with complex locality constraints on each node is adopted by Zhang et al. [11]. With the only exception of the last one, all the reported approaches rely on profiling information which are derived either from frequency profiling or by analyzing the output trace of an instruction-set simulator (ISS).

3 Memory Maps Translator

Whether code or data objects are considered, and whatever placement algorithm is adopted, the validation of new objects placement is usually done by simulation. First, a new program executable file is built so that the new layout is taken into account. Then, an ISS is used to evaluate both the new miss rate and, possibly, updated profiling information for a successive iteration of the placement algorithm (Figure 1). Depending on the case, the time effort required to do such validation may be negligible or not. However, it may clearly become considerable

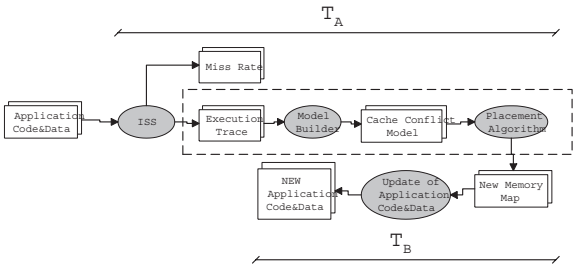

Figure 1. Typical tool flow for placement algorithms (the dashed box encloses the placement-dependent part)

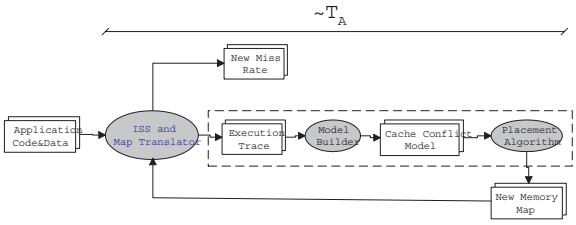

Figure 2. Improved tool flow with memory map translator (on-the-fly generation of the trace)

when several placement–validation iterations are required to obtain a satisfiable result. Building a new executable file at each iteration can be quite time consuming for applications of realistic size. And, generation and storing of huge execution traces on memory mass disks may easily dominate the overall computing time (which may even be of several hours). This is not surprising, in fact many methods have been proposed (e.g. [1]) in order to relieve such last drawback. Unfortunately none of them is completely satisfiable, so it is still important to exploit any possible improvement of the computing time.

Our approach, which is depicted in Figures 2-3, provides an improved solution. We have developed a module, named *memory map translator*, which converts on-the-fly an input trace belonging to a first memory space to an output trace which belongs to a second memory space. This permits to evaluate any code or data placement without the need of updating the application file, so practically reducing to zero the time T_B of Figure 1. Moreover, even when a trace-driven simulation approach is adopted (Figure 3), it is not necessary to generate a new execution trace after each iteration because it is sufficient to reuse the original execution trace with any new memory map. The elimination of trace

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

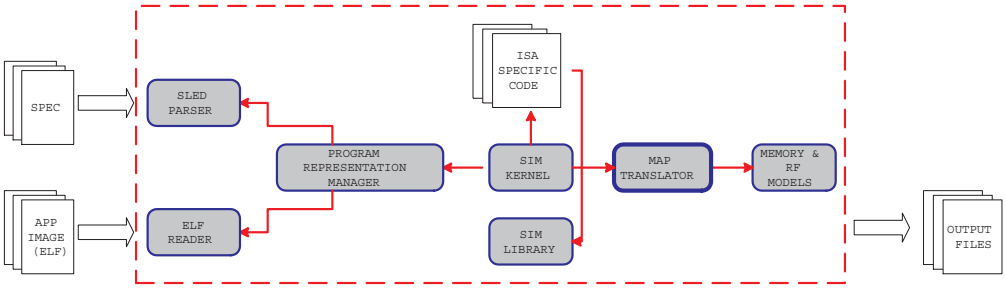

Figure 4. Structure of the RTOOLS retargetable simulator [6]

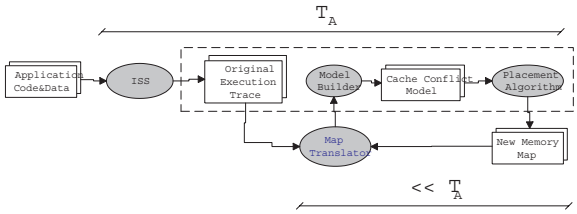

Figure 3. Improved tool flow with memory map translator (strictly trace-driven approach)

generation and storing is a clear benefit of this configuration that usually results on a much shorter computing time. An additional advantage is that the translation process is completely independent of both the adopted placement algorithm and the processor instruction-set simulated by the ISS, so resulting on a very generic and reusable solution.

We have implemented the map translator in our ISA-retargetable simulator [5, 6] as it is shown in Figure 4. Even in this case the execution trace is generated and translated on-the-fly without the need of storing any information on memory disks. Each memory reference (i.e. instruction load or data load/store) which is produced by the simulator's kernel is processed by the translator. Its translation is simply done by producing a new memory reference where the address has been translated of a given amount. All the other characteristics of the input memory reference, such as address/data lengths or their binary representation or others, are left unchanged. Such translation process can be implemented in several ways. We opted for a solution based on a table that is indexed by the request address, because this is at the same time simple and efficient enough for our purposes. Each record of the table contains two fields: a memory interval and an offset value. The translation works in two steps: first, a binary search is used to identify the interval the memory reference belongs to and, then, the related offset value is added to the memory reference. Of course, forward or backward placement will result from the sign of the offset. If the input memory reference does not belong to any interval of the table, then it is assumed an offset of value zero. That is the output reference is exactly equal to the input one. As it is shown in Appendix A this strategy guarantees a better worst-case and average-case performance than assuming to have a complete memory map where a zero value offset is associated to each object whose layout is kept unchanged.

Our solution is extremely simple and straightforward. Moreover, since it is based on the processing of input/output traces of generic objects and any generic memory map may be represented by a sorted table of address intervals, it well applies to the placement of any code or data granularity. For example, considering procedure placement algorithms [7] it is possible to build a memory map where a single interval is associated to each procedure: the interval bounds are exactly the lower and upper address of the procedure. In the case of basic blocks reordering [4] a memory interval is associated to each basic block. In general, a memory interval may be associated to each code or data object to be placed in the new layout.

As example, the following table relates to the procedure layouts of Figure 5:

Memory Interval	Offset
$[0x0100, 0x0270]$	D_1
$[0x0271, 0x0390]$	D_2
$[0x0391, 0x043d]$	D_3

The original memory layout, which is the one actually coded in the application file, consists of four procedures. Procedure P0 is the boot code and it has been placed at the absolute address 0x0 by the linker. The other three procedures constitute the actual application; their sequential placement has been done by the compiler and their relocation at the address 0x0100 by the linker. In the layout n.1, we have exchanged the order of procedure P1 and P3. The new address of P1 is became 0x02cd, resulting on an offset of +0x01cd bytes as it is reported in the previous table.

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

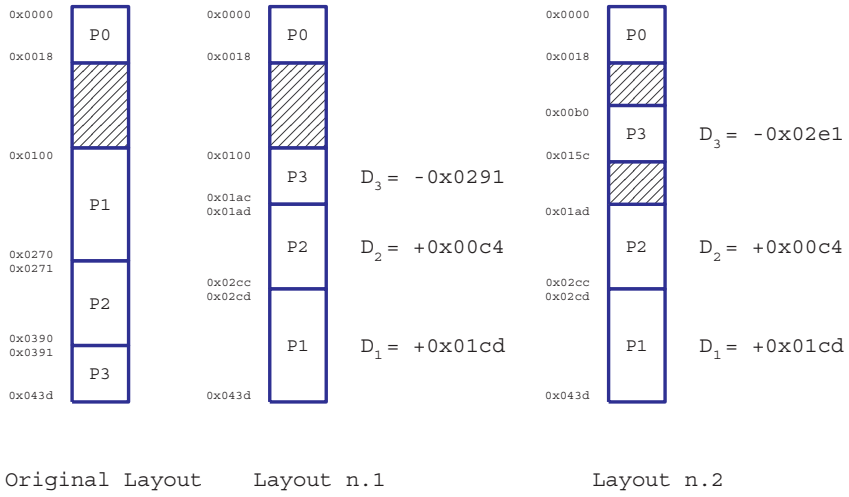

Figure 5. Original and derived memory layouts of the adpcm benchmark

Similarly, the two others records of the table refer to the procedures P2 and P3; while procedure P0 was not relocated in the new layout. The next example, layout n.2, shows a different relocation of P3 and the insertion of empty space between P3 and P2. In order to make the previous table still valid as memory map, it is sufficient to update the offset D_3.

4 Experimental Results

We have implemented the two schemes of Figures 2-3 in our ISA-retargetable simulator [5] and we have done some experiments to evaluate the simulation overhead associated to the map translator. It is extremely important for us to have a very efficient implementation because a lot of simulation time is required to tune real software application to a given hardware platform. A realistic execution of few seconds of embedded code usually consists of several hundreds millions instructions and many memory configurations need to be explored before reaching a useful performance-cost tradeoff point.

First, we have considered the ST210, an high-performance VLIW embedded processor of STMicroelectronics, which has been configured with an on-chip direct-mapped instruction cache and a set of benchmarks of medium complexity. The results are reported in Table 1. As it was expected there is a reduction of the simulation speed due to the translator overhead but the relative variation is quite small, about 4.7% in average. This is absolutely acceptable for our current practice.

Similar results were obtained by considering a different microprocessor, the ARM7, as it is reported in Table 2. Even in this case the reduction of simulation speed is less than 5% in average. The two experiments refers to different microprocessors and different cache configurations, however their experimental overhead is practically the same. This was expected because the translator implementation is completely independent from both the simulated microprocessor and the memory architecture. It obviously depends on the adopted memory map, but such dependency is quite loose.

5 Conclusions

A retargetable simulator has been enhanced with the capability of simulating on-the-fly any data or code memory map. The beauty of the provided implementation mainly relies on its joint simplicity and effectiveness. Experiments show a small simulation overhead of the memory map translator which is largely acceptable for our practical purposes. The flexibility of the proposed solution allows the adoption of many different program placement algorithms without limiting their layout granularity. Moreover, the proposed tool flows largely improve the overall computing time.

Appendix A

As it is widely reported in literature, the average and worst-case complexity of the binary-search algorithm on a sorted table with N entries is $O(log_2 N)$. We want to exploit this result to compare two different strategies for our memory map translator:

1. To adopt a *complete* memory map where the objects

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

Benchmark	Description	Exec. Instr. (Mil)	ISS Speed. (MIPS)	ISS+Trasl. Speed (MIPS)	Variation (%)
adpcm	Adaptive Pulse Code Modulation Encoder/Decoder	333	1.46	1.40	4.1
chk_data	Statistical checking of a data set	137	1.34	1.26	6.1
circle	Firmware routing to compute the point coordinates of a circle	124	0.87	0.83	4.6
des	Data Encryption Standard	877	1.35	1.24	8.1
dhry	Dhrystone C Benchmark	343	1.46	1.37	7.0
djpeg	JPEG Image Decoder	141	1.00	0.95	5.1
fft	Cooley-Tukey Fast Fourier Transform (1024 points)	32	1.02	0.97	4.6
g721_encoder	Singnals Encoding	538	1.42	1.35	5.2
edetect	Edges Detect Algorithm	793	1.39	1.35	2.9
jfdctint	Integer Fourier Discrete Cosine Transform	124	1.29	1.22	5.4
line	Graphical routing to compute the points coordinates of a line	130	1.06	0.99	6.4
matcnt	A multiplication test on arrays	124	1.36	1.32	3.1
mpeg_low	Berkeley MPEG Player with Low Quality	682	1.28	1.24	3.1
motion	MPEG2 Motion Estimation Algorithm	75	1.42	1.37	3.6
piksrt	Sorting Algorithm n.1	278	1.42	1.35	4.5
sort	Sorting Algorithm n.2	103	1.77	1.67	5.6
stats	Statistical analysis of a data set	208	0.98	0.94	3.8
tcode	Turbo Code	445	1.38	1.35	2.2
whetston	Whetstone Benchmark in C	385	0.95	0.92	3.1
Average		309	1.27	1.21	4.7

Table 1. Benchmarks description (VLIW ST210 target on a 750Mhz Sun Host)

that are not relocated have an offset equal to zero.

2. To adopt a *partial* memory map where only the relocated objects are reported. During reference translation, if an address is not found in the memory map then it is assumed an offset equal to zero.

So, let's consider a complete memory map of n objects (i.e. a table with n records {memory interval, offset}) where a fixed percentage α of such objects are not relocated (i.e. the offset is equal to zero). Let's assume that an execution trace consists of K memory references, uniformly distributed in the memory space. It follows that the execution time for the first strategy is proportional to $K * log_2(n)$. While in the other case, it is proportional to:

$$\begin{aligned}(1-\alpha) * K * log_2((1-\alpha) * n) \quad &+ \\ \alpha * K * log_2((1-\alpha) * n) \quad &= \quad K * log_2((1-\alpha) * n)\end{aligned}$$

If we consider their ratio $log_2((1-\alpha) * n)/log_2(n)$ we see that it is strictly less than 1 for $n > 1$. So, the second strategy always outperforms the first one. In particular, considering that: 1) reasonable values of n for an application of medium complexity are between 100 and 1000 when we refer to procedure placement, and they are between 1000 and 10000 if we refer to basic block reordering; 2) worst-case values of $(1-\alpha)$ may be estimated between 0.01 and 0.10 if we refer respectively to the reordering of a small number of basic blocks or procedures. It follows that the previous ratio has an estimated value between 0.4 and 0.6. That is, in the best-case, the second strategy results on a performance improvement of roughly 40%-60% which is a considerable amount.

References

[1] W. W. J. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3):222–241, 1991.

[2] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

Benchmark	Description	Exec. Instr. (Mil)	ISS Speed. (MIPS)	ISS+Trasl. Speed (MIPS)	Variation (%)
`circle`	Firmware routing to compute the point coordinates of a circle	106	1.15	1.10	4.4
`des`	Data Encryption Standard	653	1.42	1.34	5.6
`djpeg`	JPEG Image Decoder	179	0.98	0.93	5.1
`fft`	Cooley-Tukey Fast Fourier Transform (1024 points)	30	1.62	1.56	3.7
`jfdctint`	Integer Fourier Discrete Cosine Transform	102	1.56	1.49	4.5
`line`	Graphical routing to compute the points coordinates of a line	145	1.31	1.24	5.3
`motion`	MPEG2 Motion Estimation Algorithm	138	1.52	1.43	5.9
Average		193	1.37	1.30	4.9

Table 2. Benchmarks description (ARM7 target on a 750Mhz Sun Host)

[3] Y. Choi and T. Kim. Memory layout techniques for variables utilizing efficient dram access modes in embedded system design. In *Proc. of Design Automation Conference*, 2003.

[4] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, 1999.

[5] G. Luculli. An ISA-retargetable framework for embedded software analysis. In *Proc. of 10th Annual IEEE International Conference on the Engineering of Computer Based Systems*, 2003.

[6] G. Luculli. Retargetable tools for embedded systems. In *Proc. of IEEE EUROCON*, 2003.

[7] S. McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 71–79, Toronto, Ontario, Canada, June 1991.

[8] S. Parameswaran, J. Henkel, and H. Lekastas. Multiparametric improvements for embedded systems using codeplacement and address bus coding. In *Proceedings of the Asia South Pacific Design Automation Conference*, January 2003.

[9] K. Pettis and R. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, 1990.

[10] H. Tomiyama and H. Yasuura. Optimal code placement of embedded software for instruction caches. In *Proc. of European Design&Test Conference*, 1996.