Predicting Hardware Acceleration Through Object Caching in AMIDAR Processors

Stefan Döbrich and Christian Hochberger
Dresden University of Technology, Chair for Embedded Systems
01062 Dresden, Germany
{Stefan.Doebrich|Christian.Hochberger}@inf.tu-dresden.de

Abstract: Dynamically reconfigurable architectures offer the opportunity to migrate software into hardware functional units at runtime. Architectures derived from the AMIDAR model exhibit such possibilities. In previous work we have shown how to identify heavily used code sequences and have also shown that it might be interesting to synthesize hardware for a set of methods of one class and also cache the state of particular objects in the synthesized hardware. In this paper we discuss a method to identify such objects at runtime and present a heuristics to select caching candidates in order to make optimal use of the limited storage resources.

1 Introduction

Configurable Systems on a Chip (CSoC) are becoming more and more important in the embedded systems market due to their cost effectiveness and flexibility. Mask costs will be very high in the future due to the required high resolution[SIA01]. CSoCs offer the possibility to implement multi protocol/multi standard systems with a single chip and thus can be produced cost effective in large quantities. Their reconfigurable part can be used to adapt the hardware to future requirements that are unknown at the time of the initial development.

With CSoCs it will be possible to implement new IO functionality and peripherals in an existing system. It will also be possible to have peripherals to increase the performance (e.g. crypto accelerators), but the processor core itself cannot be changed or enhanced. To overcome this problem, we have introduced the AMIDAR class of processors[GH05c] which can be adapted to requirements of an application at runtime.

Although C as a programming language still dominates the development of embedded software, the growing tendency to use Java makes this language an attractive object of study. Due to the code shipping abilities of Java, it is most likely, that especially systems programmed in Java will experience a shift in the requirements during their lifetime.

Running Java bytecode on an AMIDAR processor offers the opportunity to introduce specialized hardware components that replace parts of the software and thereby enhance the performance of the system. Profile information of the running application is required to manage this dynamic hardware/software partitioning.

The processing of multimedia content becomes one of the major challenges in embedded

systems. Presentation typically requires a high amount of computing power which can be supported by specialized hardware. In [GH05b] we have shown the typical speed-ups that can be achieved with such a hardware. For these reasons we have chosen the decoding of an MPEG stream as a sample application for our measurements.

In this contribution we show a methodology to predict the effects of caching the state of objects in hardware (reducing the number of memory accesses). This takes the number and type of accesses to the objects state into account as well as the management of the coherence of the object state.

1.1 Related Work

Hardware implementations of Java bytecode processors are available in a large number. Although the JOP processor[Sch04] is a bytecode processor especially for FPGAs, its internal structure cannot be adapted at runtime. To our knowledge, only the JEM-II processor can be customized for the application requirements[aji00].

Also, customized accelerators[HHVea02] have been built to speed up the execution of Java bytecode. In this case only a small part of the bytecode execution is implemented in hardware and the main execution is done on a conventional processor.

Other researchers have addressed reconfigurable processors in general. They typically depend on compile time analysis and generate a single datapath configuration for an application beforehand[ECF96][CPSS00][HHHN98]. Very few processors are really reconfigurable at runtime[PTD99]. But even in this case, the configurations and the time of reconfiguration are defined at compile time. Thus, the tools for these architectures either do no profiling at all or have to resort to profiling the applications before evaluating the configurations and these architectures can't react on varying requirements. Also, these synthesis systems typically start from variations of the language C and thus cannot discuss any object related effects.

1.2 Paper Outline

In section 2 an overview on the AMIDAR model and the derived implementation of a Java bytecode processor are presented. In section 3 we discuss our previous results of profiling in the AMIDAR model. We show a method to evaluate the effect of object caching in section 4 and present a heuristics to determine the efficiency of caching operations in section 5. Based on this method we discuss the characteristics of a sample application in section 6. Finally, a conclusion and an outlook onto future work are given.

2 The AMIDAR Model

In this section we give an overview on the AMIDAR model of an adaptive processor as it can be seen in figure 1. It consists of four main types of components: a token generator,

functional units (FU), a token distribution and a communication network. The token generator is a specialized FU, which is always required. It controls the other components of the processor (e.g. code memory) by means of tokens. A set of such tokens is generated for each processed bytecode and is sent to the FUs over the token distribution network. The tokens tell the FUs what to do with input data and where to send the results.

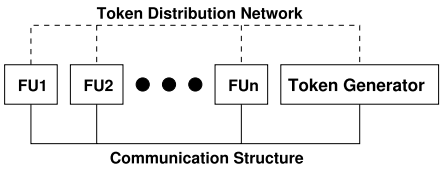

Figure 1: General model

Program information, address information and application data is passed between the FUs over the communication structure. Every distributed token and data is tagged so every operation works on the correct corresponding information.

2.1 Principle of Operation

Program information (i.e. the instructions) is sent to the token generator. It creates a set of tokens for this instruction and distributes them to the FUs. A FU starts the execution of a specific token as soon as all input data with the corresponding tag has arrived. Upon completion of an operation the result is sent to the destination. The tag that is sent together with the result optionally can be incremented. An instruction is completed, when all the corresponding tokens are executed. One of the tokens must trigger the sending of the next instruction to the token generator. This data driven approach has a number of advantages:

- The parallelism is only limited by data dependencies between consecutive instructions. These dependencies can only originate from application data.
- The execution of an instruction will work, independent of the time required by a FU to complete its token or of the structure of the communication network. Thus, it can be changed at runtime without the need to reconfigure the token generator or timing information.
- It allows overlapping execution of instructions, as the token generator can start distributing tokens for more than one instruction. Data which belongs to different bytecodes has a different tag information.
- New FUs and instructions using these FUs can be introduced (which requires an extendable token generator).

This model isn't a dataflow machine as it processes instructions and not application data. Precautions for dynamically synthesized FUs and flow control mechanisms (e.g. prevention of race conditions), can be found in[GH05c]. A description of the achieved parallelism and some figures for the efficiency (clocks per instruction) can be found in [GH05b].

2.2 Applicability

In general, the presented model can be applied to any kind of instruction processing, where a single instruction is composed of microinstructions. The model produces the best results if parallel execution is possible due to no strict order of those microinstructions and a minimum of dependencies between consecutive instructions. Overlapping execution of instructions comes automatically with this model. Intermediate virtual assembly languages like Java bytecode or the .NET code seem to be good candidates for instruction sets. The range of FU implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and basic operations are sufficiently complex.

2.3 Adaptive Operations

Due to the robust nature of the AMIDAR model it is easy to implement various adaptive operations. Firstly, it is possible to exchange FUs with other versions that provide other properties (e.g. smaller area). Also, it is relatively easy to add new FUs which can still use other FUs (e.g. code memory) but implement additional functionality. Secondly, we can change the number and design of the bus structure.

2.4 Implementation of a Java Processor

The Java bytecode processor we derived from the AMIDAR model consists of the following functional units: The *code memory* stores the Java bytecode and some additional class and method data. The Java bytecode is sent from the *code memory* to the *token generator*, which splits the opcode into several tokens. Since Java bytecode is a stack oriented language, we need an The *operand stack* stores temporary results during computations. Parameters and local variables of a Java method are stored in the *local variable* memory. The *ALU* does all computations. The *method stack* stores the call hierarchy. All runtime Java objects are created in the *object heap*. The *jump unit* is required to execute branches. The execution of an application on an AMIDAR processor without specialized hardware FUs shall be regarded as *normal execution* throughout this paper.

3 Object Oriented Profiling of Applications

In [GH05a] we have shown a methodology and circuit to acquire a fine grained profile of a running application. It allows a sensible decision on the scope of code which will be transformed into hardware. This is useful for methods which are too large to fit fully into the hardware already delivers speedup factors of more than 10. The main source for this speedup is the elimination of the operand stack (which will vanish in the datapath).

Further acceleration can only be achieved, if we elminate more memory access operations. The largest number of remaining memory accesses results from acesses to the object fields on the heap. Storing such objects or arrays in the hardware would also eliminate the corresponding memory accesses to them. As only the most recently used object of a

particular class will be kept in hardware, it will work like a caching mechanism.

Caching the objects state in hardware can significantly reduce the execution time of a hardware implemented code fragment dealing with that object. All accesses to fields of the object will come for free in that case. But there can be some additional cost of runtime and hardware too. If those costs (see section 4) are to high the resulting performance may be worse than without caching. Thus, the effect of caching a particular object should be accurately predicted in order to make good decisions about the synthesized hardware.

4 Measuring the Effect of Object Caching

In Java object variables are called fields and can either consist of one single word (e.g. int) or one double word (e.g. long). Caching those fields will improve runtime through eliminating transfers between object heap and the corresponding FU.

4.1 Access to Object Fields

The first step to evaluate those field access operations is to observe the currently active object for every loaded class in the object heap. All following field access operations related to this object are counted. When another object of the same class is activated, the current objects measurement frame ends and the number of counted accesses can be related to its duration. This speedup statistics can be generated for the whole runtime and all used objects.

4.2 Runtime of Hardware Implemented Methods

Clear predictions about an applications runtime behavior can only be made with a runtime model for all bytecodes. It's possible to give an estimation of the latency for every instruction with constant runtime. The Java instruction set can be split up into eight groups with partially different hardware clock cycles (hcc):

- arithmetical operations - one hcc
- array operations - 2 + words hcc
- conditional operations - zero hcc
- field operations - 1 + words hcc
- invoke operations - nargs hcc
- return operations - one hcc
- stack operations - zero hcc
- other operations - bytecode specific

The number of clock cycles for a field access operation depends on the actual size and type of the field as a different number of words may be handled. The group of other operations consists of rarely used bytecodes with a more complex runtime behaviour.

This leads to a conservative prediction of the speedup effects through object caching. The number of field access operations on this object have to be related to the applications runtime. Thus, the speedup through object caching on an object can be formulated as:

$$s = cyc/(cyc - 2 * swa - 3 * dwa) \quad (1)$$

In this formula cyc is the number of hardware execution cycles, swa stands for the number of single and dwa for the number of double word access operations. The formula can be used to calculate the global application speedup of a fully hardware implemented program.

This formala is conservative, since objects containing arrays are not treated appropriately. In Java bytecode arrays are handled through references. Thus, an object containing an array looks to the execution unit like an object containing a reference to another object, which normaly could not be cached. We need to extend our formula to allow basic type arrays to be cached. Arrays are accessed through load/store operations via an offset to the related entry. If an array is stored in an object, a `getfield` operation precedes every of those load/store operations.

```
public class Application {
  int[] objectArray = new int[5];

  public void main(String[] args) {
    int localInt = objectArray[0]; // (1)
    int[] localArr = objectArray;
    localInt = localArr[0];        // (2)
  }
}
```

Figure 2: Sample Java code

It seems to be simple to count those access operations, but unfortunately this way of accessing data structures generates an alias problem. A look at the code example in figure 2 and the related bytecode sequence in figure 3 will illustrate this problem.

```
(1) aload    // push objects reference
    getfield // push arrays reference
    iconst   // push array fields offset
    iaload   // push array fields value
    istore   // store value in localInt

(2) aload    // push arrays reference
    iconst   // push array fields offset
    iaload   // push array fields value
    istore   // store value in localInt
```

Figure 3: Sample bytecode sequence

In (1) the array is stored in the object and the sequence of operations can be traced easily to evaluate the number of saved memory access operations. Example (2) presents a different situation, where the reference to the array is temporarily kept in a local variable and thus, the access to the array can't be easily related to the object where it is stored.

4.3 Solving the Alias Problem

The described alias problem can be solved. The solution is based on bytecode manipulation and memory operation modification. At first it's necessary to split the `getfield` operation into two slightly different bytecodes. The first one is used to access standard single word or double word fields, the other one to access object arrays. This `getfield_array` operation leads to a special object heap operation which manipulates the pushed reference,

by setting the highest address bit. The object heaps reduction from four gigawords to two gigawords doesn't seriously limit the implementations functionality. The manipulated bit is only used for profiling purposes, and isn't regarded by any other operations.

This procedure solves the alias problem for one dimensional arrays, and the first dimension of multi-dimensional arrays too. All other dimensions are accessed via a chain of `aaload` instructions which push the reference to the next dimension onto the stack. Those references are manipulated in the same way and this solves the alias problem for higher dimensions.

The execution time of an array access operation depends on the length of the accessed field. Single word fields can be accessed in three clock cycles and double word field accesses consume 4 clock cycles. According to these accelerations the benefit through caching of objects (including arrays) can be formulated as:

$$s = cyc/(cyc - 2swa - 3dwa - 3swaa - 4dwaa) \tag{2}$$

The count of single and double word access operations is represented by $swaa$ and $dwaa$. This formula doesn't regard any costs that have to be spent for caching and decaching operations. These costs depend on the number of those operations per class, and on the size of a class' object. Every word of an object needs one clock cycle to be transfered from the object heap to the cache and vice versa. The transfer costs for a number of n classes can be written as:

$$totalcosts = \sum_{i=1}^{n} \#words_i * (insertion_i + copyback_i) \tag{3}$$

The number of caching and decaching operations can differ, because an object can be cached and doesn't need to be decached. This cost has to be regarded when the speedup is calculated:

$$s = cyc/8cyc - 2swa - 3dwa - 3swaa - 4dwaa + totalcosts) \tag{4}$$

5 Heuristics

Only limited storage resources are available for the caching of objects. Thus, a decision has to be made which objects should be cached. Such a ranking has to take into account also the hardware costs of the caching (storage elements), since if the caching of two objects has the same speedup, the smaller one makes better use of the storage elements.

5.1 Construction of a Heuristics

Our primary criterion for the quality of object caching operations shall be their efficiency. Therefore, it is necessary to regard all costs. Thus, the number of projected gained clock

cycles of a caching operation is related to the corresponding object size. This fraction then gives information about the number of gained clock cycles per cached object word and is called efficiency of a caching operation.

$$efficiency = gainedcycles/objectsize \tag{5}$$

Possible candidate objects are sorted by decreasing efficiency and as many objects as fit into the storage elements are selected from the start of the list.

5.2 A Synthetic Scenario

The displayed graph in figure 4 shows several effects which can occur according to bad predictions on hardware acceleration and continuous or frequent changes of the cached object. Field access operations to two objects of the same class have been evaluated.

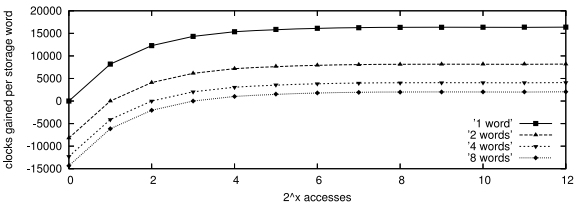

Figure 4: A set of synthetic access profiles

A basic data type field (32 bit) of both objects was accessed 2^{12} times. Each of the four curves displayed in figure 4 shows the measured data regarding to different object sizes. Every displayed measurement point within a single curve shows the gain through object caching in clock cycles depending on the exchange of the active object every 2^x access operations. It shows that wrong predictions on speedup effects can lead to increasing runtime. At the break even point the runtime costs of a caching operation are compensated by the number of gained clock cycles during the objects caching period.

6 Results

We implemented our analysis method in the AMIDAR model. In order to achieve correct results, only methods which don't contain invokes where folded to hardware, and only field access operations occurring in those methods where evaluated.

6.1 Test Scenario MPEG-Decoder

Our test application is a part of a typical MPEG processing chain. It consists of the Inverse Discrete Cosine Transformation (IDCT), where coefficients in the frequency domain are often encoded with variable bit length by means of a standardized Huffman tree. After the IDCT a color transformation is often required, for instance from YUV to RGB.
The input data for the complete processing chain is a sequence of Huffmann encoded coefficients. For each 8x8 block of pixels a new IDCT object is instantiated for each color

component. Each repetition of the MPEG chain produces one such block. During the construction of all IDCT objects a single Huffmann object is used. Each of the IDCT objects is then used to produce a matrix of 8x8 pixel color components, which eventually are transformed into RGB pixel data. All methods of both classes are free of invokes and can be folded into hardware. A Huffmann object consists of 940 words, and an IDCT object consists of 258 words. No object contains double word fields or references to arrays of double word types, so the number of access operations to those fields is zero.

blocks	i_cyc	h_cyc	swa	swaa	hc_cyc	cs	os
1	391103	83026	8373	10240	37790	2.20	10.35
2	767108	150953	16746	20480	61679	2.45	12.44
5	1895123	354734	41865	51200	133346	2.66	14.21
10	3775137	694358	83730	102400	252780	2.75	14.94

Table 1: Access Profile of the MPEG chain

Table 1 shows the result of our runtime and object access profiling. The column *swa* contains the number of single word access operations, the column *swaa* shows the number of single word array access operations. These values are accumulated for the applications whole lifetime. It is also possible to generate those statistics for every single object too.

The application runtime is given for different execution modes: *i_cyc*, normal execution (no method folding or object caching) *h_cyc*, method folding. *hc_cyc*, method folding and caching of objects. The information shown in columns *h_cyc* and *hc_cyc*is calculated with the given formulas. The column *cs* shows the folded applications speedup through object caching. The column *os* displays the applications overall speedup through method folding and object caching, related to the clock cycles of normal execution. The speedup grows from loop to loop due to initialization effects.

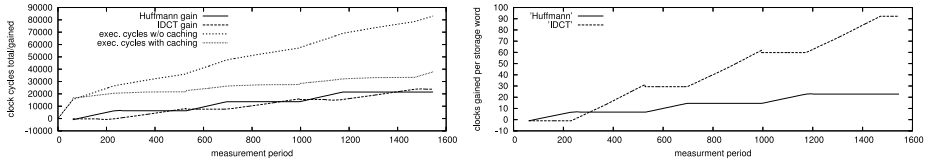

Figure 5: Execution statistics of the MPEG processing chain

6.2 Prediction At Runtime

The above discussion relates to the analysis of an application that is already finished, but it is necessary to be able to make clear predictions at runtime. Figure 5 shows the MPEG chains profiling at runtime, where the statistics values have been evaluated every 250 clock cycles. Such a continuous profile can be used for evaluation purposes at runtime. The speedup that is gained through object caching varies from point to point. The Huffmann object provides a higher gain, and therefore a better speedup most of the time. Thus, the Huffmann object would be the favored caching object in regard to the pure clock cycle gain.

However, we have shown that information about the object size has to be considered as well. Thus, it is necessary to relate the pure clock cycle gains displayed in figure 5 (left) to the size of the corresponding objects. The curves that show the number of gained clock cycle per word for both objects is displayed in figure 5 (right). Caching the active IDCT object leads to a higher efficiency than caching the Huffmann object. A better way to draw a decision could take into account not only the current measurement but previous measurement points and predict the future behavior through simple linear regression.

7 Conclusion and Future Work

In this contribution we have shown a method to gather the object access statistics for individual classes in AMIDAR processors. Furthermore, we presented a heuristics which rates projected object caching operations by their efficiency. This information allows the selection of caching candidates out of a set of objects. Future work will concentrate on an efficient hardware implementation of such a profiling mechanism.

References

[aji00] ajile Systems. aj–100 Datasheet, 2000. http://www.ajile.com/.

[CPSS00] Y. Chou, P. Pillai, H. Schmit, and H. P. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 147–158, Monterey, December 2000.

[ECF96] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, Berlin, 1996. Springer.

[GH05a] Stephan Gatzka and Christian Hochberger. Hardware Based Online Profiling in AMIDAR Processors. In *IPDPS*. IEEE Computer Society, 2005.

[GH05b] Stephan Gatzka and Christian Hochberger. On the Scope of Hardware Acceleration of Reconfigurable Processors in Mobile Devices. In Ralph H. Sprague, editor, *Proceedings of the 38th Annual HICSS*, page 299. IEEE Computer Society, 2005.

[GH05c] Stephan Gatzka and Christian Hochberger. The AMIDAR Class of Reconfigurable Processors. *Journal of Supercomputing*, pages 163–181, 2005.

[HHHN98] Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Using the KressArray for Reconfigurable Computing. In John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 150–161, Bellingham, WA, 1998. SPIE – The International Society for Optical Engineering.

[HHVea02] Yajun Ha, Radovan Hipik, Serge Vernalde, and et al. Adding Hardware Support to the HotSpot Virtual Machine for Domain Specific Applications. In M. Glesner, P. Zipf, and Renovell M., editors, *Field-Programmable Logic and Applications (LNCS 2438)*, pages 1135–1138, Berlin, Heidelberg, 2002. Springer.

[PTD99] Krishna V. Palem, Surendranath Talla, and Patrick W. Devaney. Adaptive Explicitly Parallel Instruction Computing. In John Morris, editor, *Proceedings of the 4th Australasian Computer Architecture Conference*, Singapore, 1999. Springer Verlag.

[Sch04] M. Schoeberl. Java Technology in an FPGA. In J. Becker, M. Platzner, and S. Vernalde, editors, *Field Programmable Logic and Applications, LNCS3203*, pages 917–921, Berlin, 2004. Springer.

[SIA01] SIA – Semiconductor Industry Association. The International Technology Roadmap for Semiconductors. http://www.itrs.net/, 2001.