Enabling Run-Time Memory Data Transfer Optimizations at the System Level with Automated Extraction of Embedded Software Metadata Information

Alexandros Bartzas[1], Miguel Peon-Quiros[2], Stylianos Mamagkakis[3], Francky Catthoor[3,4], Dimitrios Soudris[1], Jose M. Mendias[2]

[1] VLSI Design & Testing Center, Democritus Univ. Thrace, 67100, Xanthi, Greece. Email: ampartza@ee.duth.gr; dsoudris@ee.duth.gr
[2] DACYA, Univ. Complutense de Madrid, 28040, Madrid, Spain. Email: mikepeon@fdi.ucm.es; mendias@dacya.ucm.es
[3] IMEC vzw., Kepeldreef 75, 3001, Leuven, Belgium. Email: mamagka@imec.be; catthoor@imec.be
[4] Also Prof. at Katholieke Univ. Leuven, Belgium

Abstract— The information about the run-time behavior of software applications is crucial for enabling system level optimizations for embedded systems. This embedded *Software Metadata* information is especially important today, because several complex multi-threaded applications are mapped on the memory of a single embedded system. Each thread is triggered at run-time by different input events that can not be predicted at design-time. New methods and tools are needed to automatically profile and analyze the dynamic data access behavior of simultaneously executing threads in order to enable memory data transfer optimizations. In this paper, we propose such a method and tool which extract the necessary Software Metadata information to enable these data transfer optimizations at the system level. We assess the effectiveness of our approach with the results for five real-life software applications using seven real-life run-time input traces.

I. Introduction

In modern consumer embedded systems (e.g., PDAs) it is common for a number of applications like video-playback, VoIP and a web browser to run simultaneously. This means that the operating system has to schedule multiple threads while data are commonly shared or exchanged between them during their execution. The existence of multiple threads makes analyzing the system memory transfer behavior a complicated task, because multi-threading eventually means that memory accesses from different threads interleave in a fine-grained way. The fact that different threads are active at the same time is especially relevant for these ones that engage in an asynchronous consumer/producer behavior. *The problem is that it is not useful to analyze the memory access pattern of each thread independently from the others.* The effort to combine *Software Metadata* information about the individual behaviors, in order to "recreate" the behavior of the whole system, ignores the importance of the interaction among the threads, and most importantly, the interleaved thread execution. Therefore, in order to enable memory transfer optimizations at the system level (rather than the individual thread level), we propose to profile/analyze the whole system and not each thread individually.

Additionally, the input to the system changes frequently at run-time (e.g., according to user actions). So, the control and data behavior of each one of the threads can not be fully determined at design-time without resorting to worst-case evaluations. Therefore, the memory storage requirements of the system can not be fully characterized at design-time and dynamically allocated data types become responsible for a significant portion of the total data transfers. *This increasing dynamic behavior shows that there is an urgent need for tools that help to analyze the run-time aspect of the system without an explosion in the complexity of the analysis*. It is also important to have a standard format to represent all this *Software Metadata* information in such a way that tools from different companies or from the academia can use it and link with each other.

The memory subsystem is a significant contributor to the overall energy consumption and cycle budget in embedded systems [6, 8]. In the past years, much effort has been devoted to optimize the memory hierarchy of such systems. As a result, it is today common that energy-efficient embedded platforms have scratchpad memories and relay on the Direct Memory Access [12] (DMA) mechanism to implement the memory transfers. DMA is especially desirable in embedded systems because it can be applied not only to I/O data transfers but also to the explicit management of scratchpad memories (that are more energy efficient, if correctly controlled by software, than hardware controlled caches [17, 21]). The optimization problem that designers usually face is the manual analysis of the behavior of the system to identify the most relevant data transfers where DMA can be applied. This problem has been extensively studied for static data types but there are currently no suitable techniques for this analysis on dynamically allocated data types. The automatic identification of relevant data transfers allows the designer to insert data transfer primitives in the right places at an early design stage. *Therefore, it is important to analyze the data transfers for employment of optimization mechanisms like the ones based on Direct Memory Access [12].*

In this paper, we propose a method and a set of tools that allow an automated identification of the relevant data transfers of dynamic data types present in the software of an embedded system. This application-specific information is in effect Metadata, which characterizes the source code of the embedded software applications. This combination of source code and its extracted Metadata can then be used by embedded system designers for multiple types of optimizations on more than one hardware platform. We apply these Software Metadata extracting tools to a set of real-life applications from the network domain in order to show their analysis capabilities. Finally, in order to demonstrate that the identified Software Metadata are actually relevant for the targeted embedded platforms, we il-

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

lustrate this for DMA-based performance optimizations and we show simulation results using real-life run-time trace inputs.

In Section II the related work is described. An overview of the proposed method is presented in Section III and each of the steps are explained in Sections IV, V and VI. In Section VII, experimental results are presented to illustrate the applicability of our analysis. Finally, in Section VIII conclusions are drawn and future work is discussed.

II. Related work

Currently, in the domain of hardware platform composition tools *Hardware Metadata* are exploited with the use of IP-XACT, which is the official set of specifications of the SPIRIT consortium for hardware IP metadata and tool interfaces [1]. Additionally, the concept and exploitation of *Software Metadata* is being explored by several research projects [2, 3] which aim to produce a standard description of the characteristics of applications running on embedded platforms that can be interchanged between tools of different vendors and academia. The work presented by different groups regarding workload characterization [11], scenario identification [9] and scenario exploitation [10] is also very relevant in the context of our proposed method to extract *Software Metadata* information to enable memory transfer optimizations. The above works focus on defining the characteristics of the run-time situations that trigger specific application behavior, which has significant impact on the resource usage and data access behavior of the applications under study. We apply related concepts to theirs but we have instantiated (and extended/customized) them in the context of profiling input-dependent application behavior.

The classical design for a simple DMA device was first introduced in 1955 for the IBM SAGE computer [12] and has always been the reference design. It typically consists of a state machine and several registers to hold source and destination addresses, transfer length and status [5]. In [8, 13] DMA combined with a software prefetch mechanism exploits the *a priori* access pattern of multimedia applications. A DMA architecture for high data rate implemented on TI TMS320C6211 C6x DSP is proposed in [7]. A method for application specific DMA controller synthesis is presented in [14].

In [4] DMA engines are used to provide efficient dynamic layout of data to memories. A run-time scratchpad management is proposed in [17] where DMA engines reduce the cost of copying data on such memories. [20] present several optimizations on scratchpad memory management whereas [15] focuses in the energy-efficient usage of DMA modules. In this paper, we not only provide automatically the Software Metadata information which is relevant and can be leveraged by this type of optimizations but, additionally, we enable with the proposed Metadata information further DMA-based optimizations on data transfers triggered by run-time events on top of the aforementioned design-time optimizations.

III. Overview of the proposed software metadata extraction methodology

The methodology presented in this paper is divided in several steps (depicted in Fig. 1). The starting point is the original source code of the applications (written in C/C++). Once it is instrumented with our profiling library, the resulting system is run using the most representative input samples. The result of these steps is a file containing all the accesses to data elements performed by the applications during their execution. If the analyzed applications were multi-threaded, accesses in the log file intermingle not only for the different data types but also for all the different threads running concurrently.

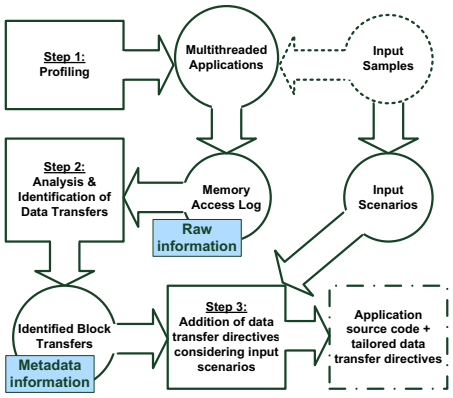

Fig. 1. Overview of the proposed methodology.

The next step is to analyze the log file to extract the information about the most relevant data transfers in the application. In this stage, several techniques are applied to differentiate among all the concurrent accesses. As a result of all the previous steps, the proposed tool produces a list with all the major data transfers ordered by *data type* and *thread*. Then, the designer can use this information to apply data transfer primitives to the sensible locations in the source code. Finally, the analysis of the behavior of the application for several relevant inputs enables the identification of different *input data instances* [9].

IV. Step 1: Creation of raw information

The first step of our approach comprises the instrumentation of the source code using a profiling library. This approach is based on our previous work in [18], where we use a set of *wrapper classes* around the dynamically allocated data types of the application. One of the strongest features of our profiling library is that it requires very small modification of the original source code: it is usually just enough to modify the variable declarations and the prototypes of the functions that use them. Moreover, as we aim to analyze multi-threaded applications, our approach is thread-safe and provides facilities to record the *ID* of the thread issuing each data access. Once the application is instrumented, the designer uses a set of relevant system inputs that can trigger the different behavior patterns that will arise at run-time during final system utilization. The applications are then fed with the different input patterns and for each execution a different log file is obtained.

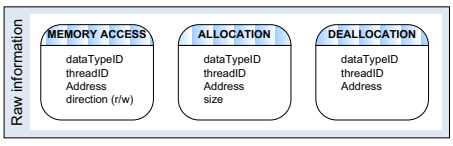

Fig. 2. Raw information contained in the log file for each data type, thread, access and storage record.

The applications are executed as it would be in the non-instrumented version although, of course, it will be sensibly slower (around two orders of magnitude according to our experiments) and will produce a big-sized log file (around tens of gigabytes in our experiments). Nevertheless, big log files are not an issue because the profiling and analysis phases can be pipe-lined, if needed, at run-time in such a way that the

log files are not actually saved to disk. The output of this step is a log file with all the raw information about the data accesses from all the threads of the application, and all the calls to dynamic memory management primitives (i.e., `malloc()` and `free()`). The entries in the log file are strictly ordered by execution time (excluding the effect of the thread-safety mechanism). Fig. 2 shows the raw information associated to each type of entry in the final log file.

V. Step 2: Extraction of Software Metadata information from raw information

The input for the second step is the log file with all the raw information of data accesses from all the threads of the application, and the calls to dynamic memory management primitives. The main task in this step is to identify the *potential* block data transfers among all the individual accesses in the log. It is important to realize that data transfer primitives have not yet been introduced on the application at this early design stage. Therefore, block transfers are not distinguished in the raw information, but their existence must be inferred from the properties of the individual data accesses. For static data types this type of analysis is well known both in the State-of-Art and in industry because the size and address of the different structures is always predefined. But for *dynamically allocated data types* it is not possible to know the actual address that correspond to each of them; *it is not even possible to know how many instances* of each one will be created at run-time.

We identify a *unique data transfer as a set of strictly consecutive data accesses to the same instance of the same data type, performed by a given thread*. With this definition, we use the following rules to identify the data transfers, briefly depicted also in Algorithm 1:

Algorithm 1 Data transfers identification

```
1:  function TransferIdentification
2:      aliveBlocks : List of Blocks
3:      recordOfTransfers : List of Transfers
4:      for all event ∈ logFile do
5:          if event is allocation then
6:              aliveBlocks.Insert(new Block(address, size, dataTypeID))
7:          else if event is deallocation then
8:              block ← aliveBlocks.FindBlock(address)
9:              if IsValid(block.activeTransfer) then
10:                 recordOfTransfers.Insert(block.activeTransfer)
11:             end if
12:             aliveBlocks.DeleteBlock(address)
13:             delete block
14:         else if event is memory access then
15:             block ← aliveBlocks.FindBlock(address)
16:             transfer ← block.activeTransfer
17:             if IsValid(block) and IsValid(transfer) then
18:                 if IsConsecutive(transfer, address, threadID, direction)
                    then
19:                     transfer.Update(address)
20:                 else
21:                     recordOfTransfers.Insert(transfer)
22:                     block.activeTransfer ←new Transfer
                          (address, dataTypeID, threadID, direction)
23:                 end if
24:             else
25:                 block.activeTransfer ←new Transfer
                      (address, dataTypeID, threadID, direction)
26:             end if
27:         end if
28:     end for
29: end function
```

1.– Independent lists are kept for alive dynamic data blocks (instances of dynamic data types), with their associated data transfers, and a record of finished data transfers. For each alive data block, the last active data transfer being performed on it is tracked and, for each active transfer, the last address accessed is also kept for the identification of consecutiveness. Fig. 3 shows the internal data structures used to extract the Metadata information from the raw information.

2.– Each time a memory allocation primitive (i.e., `malloc()`) is encountered, a new dynamic data block representation, univocally identified by its starting address and size, is created in the analyzer.

3.– For each deallocation primitive (i.e., `free()`) in the log file, the corresponding block is identified by its starting address and the corresponding representation in the analyzer is destroyed. If the analyzer was keeping track of a transfer that involved this data block, then the transfer will be considered as finished and moved to the record of transfers.

4.– For each access in the raw information of the log file, the tool analyzes if it is inside the boundaries of an alive dynamic data block by checking the access address against the starting/ending addresses of the alive blocks. Then a) if this is the case, then it checks the last transfer registered for this block (the "active transfer") and performs the consecutiveness analysis; b) if the new access is found to be consecutive, then the transfer is updated; c) otherwise, the transfer is closed and moved to the record of data transfers *IF* its length is bigger than one word and, d) finally, a new active transfer is built using as starting address the one from the data access that is currently being processed. If there is not an active transfer for the dynamic data block associated to the last data access, then a new data transfer is created. As a last possibility, if there is not an alive dynamic data block covering the address of the last data access, then it is qualified as an access to a block of static data.

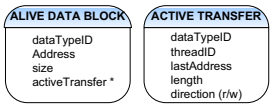

Fig. 3. Information of the analyzer internal structures.

As explained before, a non-consecutive access to a dynamic data type from a thread results in the finalization of the previous transfer for the corresponding block. The deallocation of the block also terminates the last active transfer into it, if any. Finally, transfers closed with a size of one access are discarded as individual (scalar) accesses; on the contrary, if they were comprised of more than one access, then they are added to the record of transfers for final statistics generation. When the raw information of the log file is completely analyzed, the record of data transfers is processed for statistics collection of the Software Metadata: overall numbers are calculated for each different dynamic data type, and collection of global numbers is performed. The output of this stage, i.e., the Software Metadata of the application, is presented in Fig. 4 and described in detail in Section VII.

VI. Step 3: Optimization based on extracted software metadata

Running the application on a set of different, relevant sets of input data, allows for the identification of different patterns in the transfers characteristics. This means that the software Metadata values are not fixed for an embedded system and their values vary according to the actual system input traces. For example, depending on whether the specific input activates one execution path or the other, some of the data transfers may be executed or not. Moreover, it may also be that the amount of

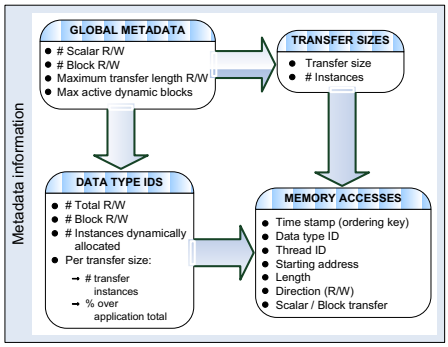

Fig. 4. Output of the analysis stage: Software Metadata information.

information that must be processed to attain the required accuracy differs dramatically from one input to other, modifying the characteristics of the memory transfers performed in the system. Our methodology allows to identify such situations and generate different solutions that make the most efficient use of resources for each case.

The use of a DMA module requires a coordinated HW/SW effort: The SW must decide whether to use or not the DMA module for any given transfer and send commands to instruct it to do the transfer. Then, the HW signals when the operation is finished (usually with an interrupt) so the SW program can synchronize while still working in other tasks during the process [12]. The decision of using the DMA for every data transfer in the system is traditionally taken at design-time for all the instances of a given transfer and subsequently hard-coded in the application source code because the designer can use only basic primitives and tools to implement the data transfers, either in software or with the help of dedicated hardware elements. But there is not currently a mechanism that allows designers to postpone the decision until the actual input data characteristics are known. Instead of this fixed approach, we propose an integrated HW-SW approach to implement data transfers, based on a design-time identification of the most relevant input data instances and detection of the one that is actually present at run-time. This means that the usage of the DMA is regulated by the software Metadata extracted by our tools at design-time and monitored at run-time.

To implement this, we propose a tailored data transfer function that will decide, at run-time, according to the detected instance of input data, how to execute the data transfer: using a DMA resource, giving or not a high priority to the transfer, performing software copies, etc. Algorithm 2 shows the pseudo-code for this function and an example of its usage. The `TransferCopy` function contains a set of rules that according to the current instance of input data and the data type that is requested to be moved, select the appropriate action to be taken (e.g., software copy, DMA transfer, etc). In the client function, the invocation of `TransferCopy` leads to implementation of the transfer policy, according to the identified case.

VII. Experimental results

In this section we present a practical usage case of our approach. In Subsection A we present the software testbench that we have used as a test driver. In Subsection B, we describe the Metadata obtained with our tools. These results are the focus of this work. Finally, as an additional verification, we present in Subsection C the results of running the memory traces ob-

Algorithm 2 Optimized data transfer function according to software Metadata.

```
1:  function TRANSFERCOPY(source, destination, dataTypeID, size)
2:      if GlobalScenario = ScenarioA then ▷ Perform a software copy
3:          memcpy(source, destination, size)
4:      else if (GlobalScenario = ScenarioB) and
            (dataTypeID = X) then ▷ Do locked DMA transfer.
5:          DMATransf(source, destination, size, WAIT)
6:      else if ... then
7:          ...
8:      end if
9:  end function
10: function PROCESSDATA(input, output, size)
11:     TransferCopy(input, SCRATCHPAD, dataType, size)
12:     DoComplexProcessing(SCRATCHPAD, size)
13:     TransferCopy(SCRATCHPAD, output, dataType, size)
14: end function
```

tained during the profiling phase in a simplified memory architecture simulator where block data transfers are performed with a DMA module. This last test is meant to verify that the data transfers identified using the Metadata are actually relevant in the used testbench.

A. Software testbench

The software testbench that we have used as a test driver is a combination of several real-life kernels present in network applications. In addition, one traffic generator was introduced to replay the network traces as if applications were reacting to user interaction and network responses. The software testbench is fully multi-threaded [19] as it is increasingly common in embedded systems: each kernel is executed in its own independent thread and communicates asynchronously with the other kernels through the use of FIFO queues. All the queues have locking mechanisms to ensure proper synchronization between threads. We have chosen this network software testbench to test the capabilities of our analysis and Metadata extraction tools because it is a good example of a data transfer dominated application. The following paragraphs describe each of the different subsystems in the software testbench.

– *User session simulator:* This kernel feeds into the testbench each packet of the input trace thus simulating user applications without their corresponding processing requirements.

– *TCP/IP packet formation:* Reflecting the entry point to the operating system like a `write()` system call, this kernel builds the complete packet filling in the header fields. This kernel then writes the new packet into the queue for encryption or the queue for TCP checksum, according to whether the connection is encrypted or not.

– *Encryption:* Packets that belong to an encrypted connection are processed with the DES algorithm.

– *TCP checksum:* The contents of the packet are thus accessed consecutively using 16-bit operations.

– *Quality-of-Service manager and Deficit Round Robin:* The QoS manager builds a prioritized list of destinations. When a packet arrives to this subsystem it is queued in one of the priority classes. Packets are extracted from them and forwarded to the network adaptor according to a simplified DRR algorithm. The forwarding of packets to the network adapter is simulated as a copy to a circular buffer in memory that can be traced by our profiling and analysis tools.

The size of the packets plays an important role to the overall memory transfer behavior. When a packet is small it takes minimal time to pass through all the stages of the testbench

framework. When a packet's size is significant, the operations of encryption and internet checksum take considerably more time. The processing time of the last kernel increases when the user sends a high number of packets to many different destinations. This fact forces the kernel to build and maintain many lists, making the task of finding the right priority queue to serve more complicated. The memory accesses that occur when a network trace triggers the aforementioned testbench are:

I.– Packets are written and deleted from the synchronization queues. When a kernel receives a packet, it is deleted from the queue it was residing. When a kernel finishes its task with a packet, it is written to the next corresponding queue. If the "destination" is full, the kernel stops until enough space is available. *The system works as a pipe-line; thus, no packet is discarded once it is accepted by the first thread.*

II.– A packet is processed by a kernel. Each kernel accesses specific packet fields, thus performing different memory accesses. The session simulator kernel just writes the raw data into the entry queue. The TCP/IP packet formation kernel adds the packet header (size of 40 Bytes) to the raw data, thus forming a packet, writing relevant values into the corresponding fields (IP addresses, port numbers, etc). The encryption kernel encrypts the data field of the packet. In order to perform that task it accesses the encryption key and relevant values from arrays residing together with the packet's data. In order to perform the CRC operation, the whole contents of the packet are accessed. The DRR kernel builds lists according to the source and destination IP addresses of the packets and then forwards the packets to the network adapter.

This system is fully multi-threaded. Therefore, memory accesses do not happen in a sequential manner. Many packets are alive at the same time during execution and memory accesses are performed at the same time from different application kernels. This is the kind of behavior that fully suits our data transfers analysis tools.

B. Data transfers analysis results

We triggered the system with a set of traces from a real wireless network. The representativeness of these traces is assured by the fact that they were obtained from different network "sniffers" in different buildings of the Dartmouth Campus and made publicly available [11]. Nonetheless, the proposed approach is applicable to any collection of real network traces.

The results produced by our analysis tool are presented here in four decreasing levels of abstraction, i.e., we start with the coarser grain information and continue entering into the details. This description corresponds with the diagram of the Metadata structure described in Fig. 4 and analyzed in Section V. The highest level of abstraction is presented in Table I. Seven input traces (first column) were used in order to trigger the application framework. The corresponding number of block transfers (reads and writes) are depicted in the second and third column respectively. The fourth column shows the maximum transfer length (in bytes). As can be seen, it is not the same for all the input traces due to the variation of the input traces themselves (variation in number of packets, average packet size, etc). The next column shows the mean length of the data transfers for each input. Finally, in the last column the ID number of the data type that is the most active is depicted. The data type that has ID 2 is the one that holds the body of the packets, whereas the one with ID 14 represents the output network buffers. The second level of abstraction is presented in Fig. 5: each representative size of data transfer (with frequency $\geq 1\%$) is presented with its frequency, after aggregating the results for all the different inputs. Table II presents the details for the concrete dynamic data type IDs associated with each transfer length (again, for frequencies $\geq 1\%$). Finally, for conciseness reasons, we do not present in this paper the raw, unaggregated data associated to each dynamic data type ID, input and transfer size. For the optimization purposes, the previous information is detailed enough but our analyzer can dump the whole unaggregated information into a text file for further processing or inspection.

TABLE I
GLOBAL STATISTICS FOR IDENTIFIED BLOCK TRANSFERS.

Input	# Read transfers	# Write transfers	Max Transfer length	Mean Transfer length	Most active data type ID
01	396,347	79,016	1,500	203	2
02	40,663	37,568	1,500	109	2
03	250,283	213,094	1,500	107	14
04	294,408	313,144	1,500	181	2
05	299,346	285,951	1,300	108	14
06	254,012	253,248	576	344	2
07	2,641,875	371,245	1,500	87	2

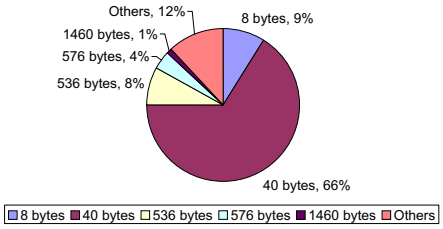

Fig. 5. Percentage distribution of data transfer lengths.

The software Metadata information produced by the analyzer is enough to allow the designer to identify the most sensitive data transfers in the application and make the most appropriate use of resources. In the following Subsection, we simulate the DMA-based optimizations to show the potential gains that can be achieved using the presented software Metadata information.

C. DMA-based optimizations using our extracted Metadata

In order to validate the relevance of the data transfers identified, we performed a simulation of the DMA-based optimizations enabled by the Metadata identification on a simple mem-

TABLE II
DISTRIBUTION OF TRANSFER SIZES FOR EACH DYNAMIC DATA TYPE ID, AVERAGED FOR ALL INPUTS.

Percentage over total	Transfer size	# Transfers	ID
55%	40	382,740	01
18%	40	126,078	14
7%	8	48,534	02
6%	536	63,210	02
5%	576	31,605	14
2%	8	16,940	01
1%	1460	6,182	02
1%	468	3,538	02
2%	Others	17,501	Others

TABLE III
REDUCTION OF THE TOTAL NUMBER OF EXECUTED CYCLES WITH DMA-BASED OPTIMIZATIONS EXPLOITING OUR EXTRACTED METADATA.

Input	# Cycles proc (with DMA)	# Cycles DMA	# Cycles (with DMA)	# Cycles (without DMA)
01	688,552	2,564,269	**3,252,821**	**4,634,128**
02	369,934	628,622	**998,556**	**1,085,164**
03	1,665,311	3,177,720	**4,843,031**	**5,607,312**
04	2,875,726	8,446,861	**11,322,587**	**15,145,873**
05	2,300,094	4,282,901	**6,582,995**	**7,527,066**
06	1,420,904	9,025,292	**10,446,196**	**16,222,819**
07	3,023,595	5,404,975	**8,428,570**	**9,775,976**
Avg.	1,763,445	4,790,091	**6,553,536**	**8,571,191**

ory hierarchy simulator. We simulate a system with one processor, one simple DMA controller, one local SRAM memory (scratchpad) and one bus to access the external DRAM memory. The simulator and the technological parameters of the memories modeled are described in [16].

To run the simulations we used the traces of memory accesses obtained from the profiling step. We substituted in the stream of memory accesses the individual ones with the block transfers that were identified during the analysis phase. Then, the simulation considers that DMA transfers block data from DRAM to the internal SRAM where they are accessed by the processor and that the processor accesses scalar data directly from the DRAM when it makes no sense to transfer them to the internal memory (no locality, accesses not belonging to any identified transfer). Table III shows the results of the simulation. For all of the inputs that we used during the original profiling, we ran both tests, using a DMA module to implement block transfers and using only a processor. For the case when the DMA is used, results are given for the number of processor, DMA and combined cycles spent in *memory bus cycles* in the system. For the case of not using the DMA, only the number of cycles spent by the processor accessing memories are presented.

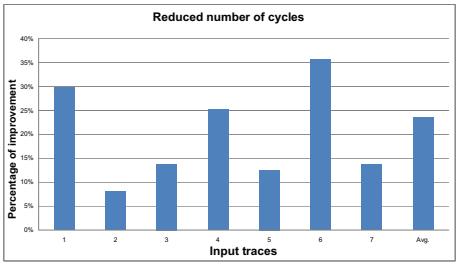

Fig. 6. Reduction (%) of total cycles spent on the bus with DMA-based optimizations exploiting our extracted Metadata.

The simulation results show that the transfers identification was successful: DMA usage allows the designer to achieve reduction of the cycles spent on the bus to fetch data, from 8% up to 35.6% (and on average 23.56% for all the traces), as depicted in Fig. 6. The overhead incurred at run-time by our approach and the dependency of the results on the actual inputs are shown in our previous work [16]. The applicability of the metadata extraction mechanism presented in this paper is, however independent of the input data.

VIII. CONCLUSIONS AND FUTURE WORK

Embedded systems with multi-threading characteristics are becoming more and more prevalent. It is not possible to analyze their memory access behavior fully at design-time. We have employed a profiling approach to gain insight on how a multi-threaded application framework (consisted of network kernels) is accessing data, and more specifically the dynamically allocated ones. We have also introduced a number of tools (profiling of memory accesses, identification of relevant data transfers) to assist the designer in the identification of the most relevant data transfers, which are represented as Metadata information extracted from the embedded software. This Metadata information is used by DMA-based optimization tools to increase the efficiency of the memory subsystem.

Given the high number of data transfers performed by the applications, a concise way is needed to represent their concurrency. As a future extension of the work presented in this paper, we are looking for a representation of this additional type of Metadata information that will help the designer in the process of scheduling the execution of the data transfers.

ACKNOWLEDGEMENTS

This work is partially supported by the PENED 03ED593 Grant from GSRT, the Spanish Government Research Grant TIN 2005-5619 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031.

REFERENCES

[1] The SPIRIT Consortium. In *http://www.spiritconsortium.org/tech/p1685/*.

[2] http://www.elis.ugent.be/resume.

[3] http://www.hitech-projects.com/euprojects/betsy.

[4] M. Absar, F. Polletti, P. Marchal, F. Catthoor, and L. Benini. Fast and power-efficient dynamic data-layout with dma-capable memories. In *1st Int'l Wksp on Power-Aware Real-Time Computing*, 2004.

[5] M. Blumrich, C. Dubnicki, E. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proc. of HPCA*, 1996.

[6] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology, exploration of memory organization for embedded multimedia system design*. Kluwer, 1998.

[7] D. Comisky and C. Fuoco. A scalable high-performance dma architecture for dsp applications. In *Proc. of ICCD*. IEEE Comp. Soc., 2000.

[8] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis. A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE T-VLSI*, pages 279–291, 2006.

[9] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 2003.

[10] S. Gheorghita, T. Basten, and H. Corporaal. Intra-task scenario-aware voltage scheduling. In *Proc. of CASES*. ACM Press, 2005.

[11] T. Henderson, D. Kotz, and I. Abyzov. The changing usage of a mature campus-wide wireless network. In *Proc. of MobiCom*, 2004.

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[13] D. Kim, R. Managuli, and Y. Kim. Data cache and direct memory access in programming mediaprocessors. *IEEE Micro*, 2001.

[14] M. O'Nils and A. Jantsch. Synthesis of dma controllers from architecture independent descriptions of hw/sw communication protocols. In *Proc. of the 12th Intl' Conf. on VLSI Design*, 1999.

[15] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-aware memory energy management. *The 12th Int'l Symp. HPCA*, pages 133–144, 2006.

[16] M. Peon-Quiros, A. Bartzas, S. Mamagkakis, F. Catthoor, J. M. Mendias, and D. Soudris. Direct memory access optimization in wireless terminals for reduced memory latency and energy consumption. In *Proc. of PATMOS*, 2007.

[17] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proc. of DAC*, 2004.

[18] C. Poucet, D. Atienza, and F. Catthoor. Template-based semi-automatic profiling of multimedia applications. In *Proc. of ICME*, 2006.

[19] Technical Committee on Operating Systems and Application Environments of the IEEE Comp. Soc. IEEE std 1003.1c-1996. 1996.

[20] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. of CASES*. ACM Press, 2003.

[21] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proc. of DATE*, page 21264. IEEE Comp. Soc., 2004.