Cache Aware Mapping of Streaming Applications on a Multiprocessor System-on-Chip

Arno Moonen[1], Marco Bekooij[2], René van den Berg[2], Jef van Meerbergen[1,3]
[1] University of Technology, Eindhoven, The Netherlands
[2] NXP Semiconductors, The Netherlands
[3] Philips Research, Eindhoven, The Netherlands
A.J.M.Moonen@tue.nl

Abstract. *Efficient use of the memory hierarchy is critical for achieving high performance in a multiprocessor system-on-chip. An external memory that is shared between processors is a bottleneck in current and future systems. Cache misses and a large cache miss penalty contribute to a low processor utilisation. In this paper, we describe a novel cache optimisation technique to reduce instruction and data cache misses for streaming applications. The instruction and data locality are improved by executing a task multiple times before moving to the next task. Furthermore, we introduce a dataflow model that is used to trade-off the number of cache misses against end-to-end latency and memory usage. For our industrial application, which is a Digital Radio Mondiale receiver, the number of cache misses is reduced with a factor 4.2.*

1. Introduction

Embedded multi-media applications are for performance and power-efficiency reasons implemented on a multiprocessor system-on-chip. External memory is required because the memory footprint of the software is considered too expensive to store in an on-chip memory. As the gap between processor and memory performance is still increasing [6], efficient use of the memory hierarchy is critical for achieving a high performance.

The number of processor stall cycles is determined by the number of cache misses and the cache miss penalty [6]. Latency in the communication infrastructure, the gap between processor and memory speed, and contention at the memory port contribute towards an increase of the cache miss penalty. A lower number of cache misses can compensate for a larger miss penalty. Furthermore, it decreases the average number of latency critical external memory accesses and thereby indirectly reduces the cache miss penalty for other processors in the multiprocessor system. Therefore, the average number of processor stalls is reduced and the system performance increases.

We focus on the class of streaming applications, which are common in the embedded domain. Streaming applications comprise a broad spectrum of applications, including audio, video, and communication processing. It is natural to represent these applications as a Cyclo Static Dataflow (CSDF) [1] graph, in which each task or component is represented by a node, which we refer to as an actor. Communication between actors is made explicit via FIFO channels, represented by the edges in the CSDF graph. For this class of streaming applications we apply the cache aware optimisation technique *execution scaling* [13], which is a transformation that improves instruction and data locality by executing each actor multiple times before moving to the next actor. If an actor is executed in a loop repeatedly, then, ideally the first iteration brings its code into the cache and subsequent iterations execute from the cache, rather then requiring it to be reloaded from memory each execution.

Disadvantages of execution scaling are (i) increase of end-to-end latency and (ii) increase of FIFO buffer capacities. The end-to-end latency increases because we execute an actor multiple times before moving to the next actor, therefore, it takes more time before the data is rippled through the CSDF graph. This problem is not severe, as many streaming applications can tolerate additional latency. The FIFO buffer capacity increases, because when executing an actor multiple times, we need sufficient capacity to store the data communicated between the actors. We describe how large FIFO buffers can be stored in the external memory and how data can be prefetched.

In this paper, we minimise the number of instruction and data cache misses by maximising the number of successive executions of an actor, while still satisfying the end-to-end latency of our application and the memory constraints of our multiprocessor system. The cache aware optimisation technique is based on execution scaling, but we target a multiprocessor architecture instead of a single processor and use uncached local (scratchpad) memories to store the input and output data of an actor. This allows us to scale the execution extensively and still reduce data cache misses. Furthermore, we introduce an algorithm to model execution scaling in a CSDF graph, such that we can use traditional dataflow analysis techniques in a design flow that maximises the execution scaling factor.

1.1. Motivating example and outline

In this section, we map a general application on a multiprocessor to illustrate the trade-off between mapping of actors to processors and the maximum allowed number of successive actor executions. Mapping consists of binding actors to processors and scheduling actors on a processor.

The general application is depicted in Fig. 1 and it has a minimum throughput constraint of $^1/_{2T}$. The actors v_1 through v_4 communicate via FIFO buffers f_1 through f_3. The actors are executed on two identical processors $p1$ and

978-3-9810801-3-1/DATE08 © 2008 EDAA

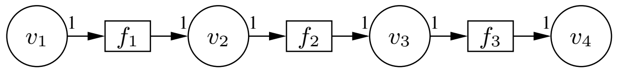

Figure 1. Example application

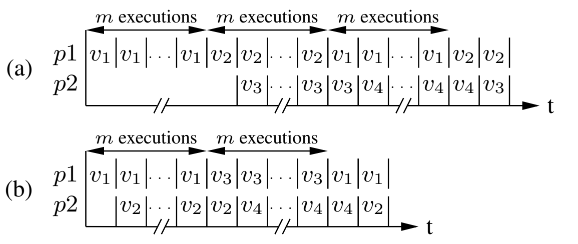

Figure 2. Two mapping options (a) and (b)

$p2$. The execution time of each actor is T time units. On each processor p, we execute two actors in a static-order schedule S_p. The static-order schedule $S_p^m = (v_i^m, v_j^m)$ represents m executions of actor v_i followed by m executions of actor v_j. Our goal is to find the mapping with the maximum execution scaling factor m that satisfies end-to-end latency and memory constraints.

In Fig. 2, we show two mapping options that satisfy the minimum throughput constraint $1/2T$. In option (a), we execute actors v_1 and v_2 on processor $p1$ and we execute actors v_3 and v_4 on processor $p2$. This mapping option requires a FIFO buffer capacity of m, 2, and m data elements for FIFO buffers f_1, f_2, and f_3, respectively. The end-to-end latency (from actor v_1 until v_4) is equal to $(2m+2) \cdot T$ time units. In option (b), we execute actors v_1 and v_3 on processor $p1$ and we execute actors v_2 and v_4 on processor $p2$. This mapping option requires a FIFO buffer capacity of 2, m, and 2 data elements for FIFO buffers f_1, f_2, and f_3, respectively. The end-to-end latency is equal to $(m+2) \cdot T$ time units.

In option (a), the end-to-end latency and FIFO buffer capacity grow with a factor $2m$ whereas in option (b) these grow with a factor m. Therefore, we conclude that mapping option (b) allows a higher value of m for the same end-to-end latency and memory constraint. A higher value of m results in less cache misses, and is hence a better mapping option. This example shows that the mapping of tasks to processors influences the maximum scaling factor m. We need tools to compute buffer capacities and end-to-end latencies for exploring different mapping options with different execution scaling factors.

The paper is organised as follows. Section 2 presents the state of the art in cache miss reduction techniques. Execution scaling is described in Section 3, and we introduce our CSDF model, in which execution scaling is modelled, in Section 4. Section 5 presents experimental results and Section 6 concludes the paper.

2. Related work

There is a large body of literature on reducing the number of cache misses, which should be applied before exploring execution scaling. First of all the cache parameters (e.g. cache line size, cache size, and associativity) have an impact of the number of cache misses [6]. Next, there are many compiler optimisations techniques for reducing the instruction and data cache misses [10]. The compiler can reduce the number of instruction cache misses by placing functions near to their callers in memory (assuming routines and callers are temporally close to each other), and by removing infrequently executed code (such as error handling) out of the main body of the code and straightening the code, so that in general, a higher fraction of the instructions fetched into the instruction cache are actually executed. For programs that manipulate large arrays of data, the number of data cache misses can be reduced by loop transformations. Examples of loop transformations are interchanging two nested loops, reversing the order in which a loop's iterations are performed, and fusing two loop bodies together into one. Cache miss reduction comes from a better use of the memory hierarchy. Execution scaling is related to loop transformations that concentrate on optimising the use of data caches, but execution scaling is focussed on transforming the main loop (scheduling of actors), whereas conventional compiler loop transformations are quite locally applied.

In the context of Synchronous Data-Flow (SDF) graphs, which is a subset of CSDF graphs [1], there is a large body of literature on scheduling these graphs to optimise various metrics. The number of context-switches is minimised in [12]. First, they use a single appearance schedule in which each task appears once and is activated a minimum number of times. Second, they scale this schedule with constraints on end-to-end latency and memory usage. The focus is a single processor with local memory and the goal is to reduce context-switching overhead cost and maximise the degree of vector processing opportunity. The number of cache misses are minimised in [7, 13] in the context of a single processor. They store the input and output FIFO buffers in a cached memory, creating the problem that the input and output data eventually overflows the data cache, when actor executions are scaled excessively.

In our paper, the focus is on mapping of CSDF graphs onto a multiprocessor architecture instead of a single processor. The input and output FIFO buffers are stored in an uncached memory region and not in a cached memory region as in [7, 13]. Therefore, input and output data cannot overflow the data cache, and execution scaling is only limited by end-to-end latency and memory constraints. FIFO buffers can be distributed between the local and external memory allowing us to create large buffer capacities. Furthermore, we present a CSDF model in which we model the application that is mapped onto a multiprocessor system with a certain execution scaling factor m. From this model, we compute the end-to-end latency and memory usage by making use of traditional dataflow analysis techniques.

3. Execution scaling

In this section, we describe our multiprocessor architecture and the execution scaling technique that minimises the number of instruction and data cache misses.

We use a tiled multiprocessor architecture and each tile consists of a processor with an uncached scratchpad memory, referred to as the local memory. The processors have level one caches for instruction and data to hide the latency in accessing the external memory. In the CSDF model of computation, two actors communicate via explicit FIFO channels, one actor producing data in the channel and one consuming the data. The FIFO channels are implemented via FIFO buffers located in the local memory of the consuming processor. A FIFO buffer is implemented as a circular buffer [3], in such a way that memory consistency is guaranteed. The processor, on which the producing actor is executed, writes the output data via the communication infrastructure into this circular buffer. If the execution scaling factor m increases, the FIFO buffer capacity also increases. This cannot lead in an overload of the data cache, since the local memory is uncached. When the FIFO buffer capacity becomes too large to store in the local memory, we distribute the buffer between the local and external memory. In our multiprocessor, we have a communication assist [2, 8], which is an automated DMA controller that prefetches the data from a circular buffer located in the external memory to the circular buffer located in the local memory. The consuming processor reads its input data from the circular buffer located in its local memory. Prefetching of data is latency tolerant instead of latency critical, as in the case when input and output data are stored in a cached memory region such as in [7, 13]. Therefore, the external memory controller has more scheduling freedom to reduce the latency of latency critical memory accesses.

Key for this architecture is that (i) we don't communicate the input and output data via the cache (preventing that execution scaling results in a data cache overflow), (ii) that we can distribute a FIFO buffer between the local and external memory, and (iii) that we use latency tolerant memory accesses.

To explain execution scaling, we use the following terminology. Let V_p be a set of actors executed on a processor p and S_p a static-order schedule with length N. The schedule is denoted by $S_p = (s_0, s_1, ..., s_{N-1})$ with $s_i \in V_p$.

After executing schedule S_p, the number of cache misses follow the line in Fig. 3, which is also observed by [6, 4]. When the cache size is small compared to size of the set of actors V_p and the cache size q increases, then the number of cache misses decrease with $\sqrt{q_0/q}$ (first order estimate), where q_0 is application dependent. If the cache size exceeds the size of the set of actors V_p, then only compulsory misses [6] (cold start misses) remain, because in our architecture the input and output data are stored in the uncached local memory. The number of cache misses can be reduced by executing an actor s_i multiple times before moving to the next actor s_{i+1} in the schedule S_p. Scaling the execution with factor m means that each actor s_i is executed

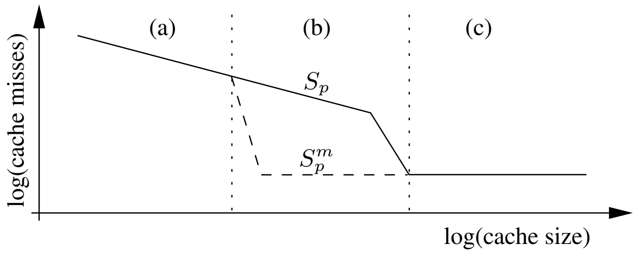

Figure 3. Cache misses as function of the cache size

m times before moving to the next actor. We refer to the new schedule by $S_p^m = (s_0^m, s_1^m, ..., s_{N-1}^m)$. After executing schedule S_p^m, the number of cache misses follow the dashed line in Fig. 3.

The impact of execution scaling on the number of cache misses for the cache size ranges (a), (b), and (c), in Fig. 3, are the following: (a) Hardly any impact on the number of cache misses, none of the actors $v_i \in V_p$ fit in the cache. (b) Largest impact on the number of cache misses. Individual actors $v_i \in V_p$ fit in the cache while the set of actors V_p does not fit. During the first execution of an actor we see compulsory misses, because the actor is being discarded from the cache when executing the other actors in the schedule. During the following $m - 1$ executions, the actor typically executes from the cache, because the program code and data is already present in the cache. The average number of cache misses reduces when increasing the scaling factor m. (c) No impact on the number of cache misses. For both schedules S_p and S_p^m only compulsory misses remain, because the individual actors $v_i \in V_p$, as well as the set of actors V_p, fit in the cache.

The more actors that are executed on the processor (i.e. the larger the set of actors V_p), the larger the size of range (b). For example, if two actors with the same size are executed on a processor, then for schedule S_p^m the flat line in Fig. 3 starts at half the cache size compared to schedule S_p. When four actors with the same size are executed on the processor, then for schedule S_p^m the flat line starts at 1/4 of the cache size compared to schedule S_p.

There are limitations to what extent the execution scaling factor m can be increased. First, it is limited by the constraints on end-to-end latency and memory usage. Second, if two actors v_k and v_l are executed on one processor, and there is a feedback loop (cycle in the CSDF graph) between these actors, then the maximum value of m is limited because of the cyclic dependency between actors v_k and v_l. The latter can be solved by executing actors v_k and v_l on separate processors, but the actors have to wait for each other due to the cyclic dependency, effecting the processor performance.

The model described in this section holds for instruction and data cache misses, because in our architecture the input and output FIFO buffers are stored in uncached memory regions.

4. Cyclo Static Dataflow model

In this section, we introduce a technique to model an application with a specific mapping and execution scaling factor. This model is used in a design flow to minimise the number of cache misses by maximising the execution scaling factor m, while satisfying the end-to-end latency and memory constraints.

The design flow is as follows. For a specified binding, initial schedules, and specific value of m, we construct a CSDF graph, as we will describe in Section 4.2. We use traditional dataflow analysis techniques for computing buffer capacities and end-to-end latency. The buffer capacities can be computed from the constructed CSDF model in combination with a given minimum throughput constraint. After computing the buffer capacities the end-to-end latency is computed. We repeat this procedure for different execution scaling factors m until we find the maximum value m that satisfies the end-to-end latency and memory constraints. Furthermore, we can backtrack for different initial schedules and different bindings of tasks to processors.

For SDF graphs, which are a subset of CSDF graphs, there is an algorithm for computing buffer capacities [15], and there is an algorithm for computing end-to-end latency [5]. Furthermore, a latency constraint can also be represented in terms of a throughput constraint [9]. Although these algorithms are intended for SDF graphs, these techniques can be extended towards CSDF graphs. If runtime of these algorithms is problematic, then a conservative approximation technique [16] can be applied to compute sufficiently large buffer capacities and an upper-bound on the end-to-end latency.

4.1. Cyclo Static Dataflow graph

A CSDF [1] graph $G = (V, E)$ is a directed graph that consists of a finite set of actors V, and a finite set of directed edges E. Actors synchronise by communicating tokens over edges that represent queues. A token can be seen as a container in which a fixed amount of data can be stored. An actor $v_i \in V$ has $\theta(v_i)$ distinct phases of execution and transitions from phase to phase in a cyclic fashion. The phase f of actor v_i in firing k is $f = ((k-1)\%\theta(v_i)) + 1$, where $x\%y$ stands for x modulo y with the result the same sign as the divisor. An actor is enabled to fire when a firing rule is satisfied, i.e. the number of tokens that will be consumed is available on each input edge. The number of tokens consumed by actor v_i equals $\gamma(e, f)$, and is determined by the edge $e \in E$ and the current phase f of the actor. The specified number of tokens is consumed in an atomic action from all input edges when the actor is started. The execution time $\rho(v_i, f)$ is the difference between the finish and the start time of phase f of actor v_i. When actor v_i finishes, it produces the specified number of tokens on each output edge $e = (v_i, v_j)$ in an atomic action. The number of tokens produced in a phase is denoted by $\pi(e, f)$. In this paper, we assume that each actor has a self-cycle $e = (v_i, v_i)$ with one initial token to exclude auto concurrency.

In a CSDF graph, the depth of a FIFO channel is theoretically unlimited, whereas in the implementation a FIFO buffer has a bounded capacity. Such a FIFO buffer can be modelled with two edges in opposite direction (a forward and backward edge). The availability of data in the FIFO buffer corresponds with the presence of tokens on the forward edge. If an actor consumes a token, it creates space in a FIFO buffer, corresponding to the production of a token on the backward edge. The number of initial tokens on both edges represents the FIFO buffer capacity.

4.2. Modelling execution scaling in CSDF

In this section, we introduce an algorithm to extend the CSDF graph representing the application into a CSDF graph modelling a specific mapping and execution scaling factor. The input of our algorithm is (i) a CSDF graph $G = (V, E)$ representing the application, (ii) a binding of actors to processors, (iii) an initial schedule S_p for each processor p, and (iv) an execution scaling factor m. The output is a CSDF graph $G' = (V', E')$ in which we model the specified mapping with schedule S_p^m for each processor p.

In this paper, we use the following terminology. Each actor is executed on a processor p and each processor p is executing actors in a static-order schedule $S_p = (s_0, s_1, ..., s_{N-1})$, with $s_i \in V$. The number of occurrences of actor v_i in schedule S_p equals $\Omega(v_i, S_p)$. For a certain schedule S_p, the k'th occurrence of actor v_i is at position $\phi(k, v_i, S_p)$, with $1 \leq k \leq \Omega(v_i, S_p)$. For the algorithm described below, we limit us to the case where two actors are executed on one processor, although the technique is applicable for more than two actors.

The new graph G' is constructed by (i) creating the new set of actors V' and (ii) creating the new set of edges E' including the production and consumption rates. (i) The new set of actors V' consists of an equal number of actors as in set V. Each actor $v_i' \in V'$ of graph G' is representing actor $v_i \in V$ of the original graph G. The number of phases $\theta(v_i')$ of actor v_i' is equal to the least common multiple (lcm) of $\Omega(v_i', S_p^m)$ and the number of phases of actor v_i, i.e. $\theta(v_i') = \mathrm{lcm}(\Omega(v_i', S_p^m), \theta(v_i))$. With this number of phases we can express the cyclo-static behaviour of the application as well as the cyclo-static behaviour of the static-order schedule. The execution time of actor v_i' can be calculated from the execution time of actor v_i and the actor switching overhead cost C_i (e.g. processor stall cycles due to cache refills). The execution time of actor v_i' in phase f is computed with Eq. (1), where $\kappa(f)$ is a short hand notation for $\phi((f\%\Omega(v_i', S_p^m)) + 1, v_i', S_p^m)$. We only have to account for the switching overhead cost if the current actor is different from the previous actor in the schedule, i.e. if $\kappa(f) \neq \kappa(f-1) + 1$.

$$\rho(v_i', f) = \begin{cases} \rho(v_i, f\%\theta(v_i)) & \text{if } \kappa(f) = \kappa(f-1)+1 \\ \rho(v_i, f\%\theta(v_i)) + C_i & \text{if } \kappa(f) \neq \kappa(f-1)+1 \end{cases} \tag{1}$$

(ii) The new set of edges E' consists of the set of edges E_b' modelling the FIFO buffers (with forward and backward edges) and a set of edges E_s' modelling the scheduling de-

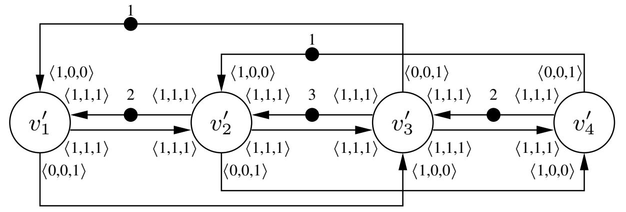

Figure 4. CSDF graph modelling the application in Fig. 1 with mapping option (b) and scaling factor $m = 3$

pendencies. The set of edges E'_b consists of an equal number of edges as in set E. Each edge $e'_b \in E'_b$ of graph G' is representing edge $e \in E$ of the original graph G. The number of tokens consumed and produced by actor v'_i on edge $e'_b \in E'_b$ equals, respectively, Eq. (2) and Eq. (3) for every phase f.

$$\gamma(e'_b, f) = \gamma(e, f\%\theta(v_i)) \qquad (2)$$

$$\pi(e'_b, f) = \pi(e, f\%\theta(v_i)) \qquad (3)$$

The static-order schedule S^m_p on each processor p is modelled with the set of edges E'_s. Each processor is executing two actors. Between these actors we add two edges in opposite directions. For actor s_0 in each schedule S^m_p, we add one initial token on the input edge $e'_s \in E'_s$. These initial tokens define which actors that can start executing. For every phase f, the number of tokens consumed and produced by actor v'_i on edge $e'_s \in E'_s$ is computed by Eq. (4) and Eq. (5), respectively.

$$\gamma(e', f) = \begin{cases} 0 & \text{if } \kappa(f) = \kappa(f-1)+1 \\ 1 & \text{if } \kappa(f) \neq \kappa(f-1)+1 \end{cases} \qquad (4)$$

$$\pi(e', f) = \begin{cases} 0 & \text{if } \kappa(f) = \kappa(f+1)-1 \\ 1 & \text{if } \kappa(f) \neq \kappa(f+1)-1 \end{cases} \qquad (5)$$

We take the application in Section 1.1 as an example to model execution scaling in CSDF. We assume the binding and static-order schedule as defined by mapping option (b) in Fig. 2. Furthermore, we assume an execution scaling factor $m = 3$ and an actor switching overhead cost C. Fig. 4 shows the CSDF model in which the schedules $S^3_{p1} = (v^3_1, v^3_3)$ and $S^3_{p2} = (v^3_2, v^3_4)$ are modelled. The number of phases of the new actors v'_1 through v'_4 equal $\text{lcm}(1, 3) = 3$. The execution times of actor v'_1 through v'_4 are $\langle T + C, T, T\rangle$. The FIFO buffers f_1 through f_3 are modelled with the forward and backward edges between the actors. The numbers beside the black dots indicate the number of initial tokens that model the FIFO buffer capacities. The input and output rates on these edges $e'_b \in E'_b$ equal $\langle 1, 1, 1\rangle$. On the remaining edges $e'_s \in E'_s$, which represent the scheduling dependencies, the input rates $\gamma(e'_s)$ are $\langle 1, 0, 0\rangle$ and the output rates $\pi(e'_s)$ are $\langle 0, 0, 1\rangle$. The two initial tokens on the input edges $e'_s \in E'_s$ from actor v'_1 and v'_2 make sure that these actors start executing in the schedules S^3_{p1} and S^3_{p2}.

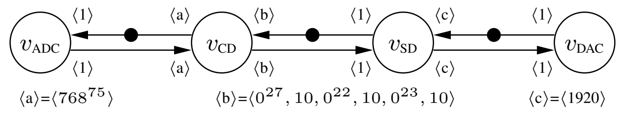

Figure 5. CSDF model of the digital radio receiver

5. Experiments

In this section we apply the cache miss reduction technique to our Digital Radio Mondiale [14] receiver. We measure the impact of execution scaling on the number of cache misses for different cache sizes and for different values of execution scaling factor m. Finally, we compute, by means of our CSDF model, the maximum value m that still meets our end-to-end latency and memory constraints.

The CSDF graph that represents our receiver is depicted in Fig. 5. The graph consists of four actors that model an Analog-to-Digital Converter (v_{ADC}), Channel Decoder (v_{CD}), Source Decoder (v_{SD}), and Digital-to-Analog Converter (v_{DAC}). The analog-to-digital and digital-to-analog converters are implemented as separate tiles in our multiprocessor system. The actors v_{CD} and v_{SD} are executed on a TM2270 which belongs to the TriMedia family [11]. We refer to this processor as the Digital Signal Processor (DSP). An external memory is applied because the code size plus the private data of v_{CD} and v_{SD} are considered too expensive to store in an on-chip memory. During our measurements, the static-order schedule on the DSP processor is $S^m_{\text{DSP}} = (v^m_{\text{CD}}, v^m_{\text{CD}}, v^m_{\text{CD}}, v^m_{\text{SD}}, v^m_{\text{CD}}, v^m_{\text{CD}}, v^m_{\text{SD}})$. We used the preamble $P_{\text{DSP}} = (v^{28}_{\text{CD}})$ before executing schedule S^m_{DSP}, in such a way that there are ten initial tokens on the edge $(v_{\text{CD}}, v_{\text{SD}})$ and actor v_{SD} is able to execute. The number of cache misses presented in this paper are measured in a SystemC [17] simulation environment that is cycle accurate.

For different instruction and data cache sizes, we measure the number of cache misses for an execution scaling factor $m = 1$ and $m = 100$, as shown in Fig. 6. The number of cache misses are measured during hundred executions of the schedule S^1_{DSP} and one execution of the schedule S^{100}_{DSP}. The cache misses in Fig. 6 follow the same pattern as the cache misses in Fig. 3. The impact of execution scaling on the number of cache misses is the largest for an instruction and data cache size of 128KByte and 512KByte, respectively. For these cache sizes, the numbers of cache misses are reduced by a factor 22.7 and 8.5 for the instruction and data cache, respectively. For smaller cache sizes the program code and private data of the individual actors does not fit in the cache, hence execution scaling has a small or no impact on the number of cache misses. When the instruction and data cache sizes grow, both actors v_{CD} and v_{SD} fit in the cache, therefore, the impact of execution scaling on the number of cache misses reduces again.

For an instruction and data cache size equal to 128KByte and 512KByte, respectively, we measured the impact of the execution scaling factor m on the number of cache misses. From this measurement we conclude that the number of instruction and data cache misses reduce when increasing the

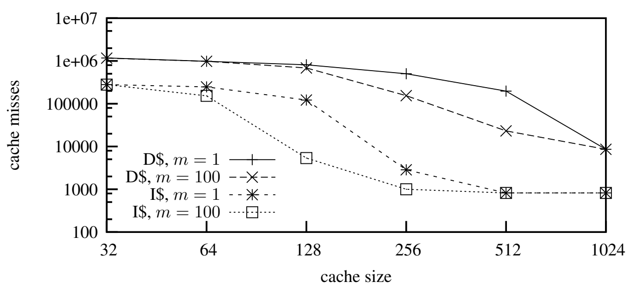

Figure 6. Instruction (I$) and data (D$) cache misses

scaling factor m. On a log-log plot (which is not shown due to a lack of space), the measured points form a straight line as expected. In the first iteration we observe cache misses, but subsequent $m - 1$ iterations we generally do not.

Finally, we compute the maximum value m that still meets our end-to-end latency and memory constraints. The throughput of our receiver is determined by the analog-to-digital and digital-to-analog converters, which have a sample rate of 48KHz. The end-to-end latency should not exceed one second. The memory usage is not critical because the FIFO buffers can be stored in the external memory, which is in the order of mega bytes. The end-to-end latency is defined as the difference between finishing the first execution of v_{DAC} and starting the first execution of v_{ADC}. An estimate on the execution time of actor v_{CD} is $\langle 2896^{27}, 14071, 2896^{22}, 14071, 2896^{23}, 14071\rangle$ microseconds and the actor switching cost C_{CD} is 631 microseconds. An estimate on the execution time of actor v_{SD} is $\langle 2202\rangle$ microseconds and the actor switching cost C_{SD} is 595 microseconds. These estimates are based on the DSP processor with a clock frequency of 300MHz, an instruction cache of 128KByte, an data cache of 512KByte, and assuming cache miss penalties of 100 and 150 DSP clock cycles for, respectively, an instruction and data cache miss. The execution times of actor v_{ADC} and v_{DAC} are equal to $1/48\text{KHz}$. For different execution scaling factors m, we derived a CSDF model via the algorithm described in Section 4.2. From this model we first computed the FIFO buffer capacities and consecutively the end-to-end latency via dataflow analysis. The end-to-end latency and the sum of the individual FIFO buffer capacities are shown in Table 1. The presented latencies include the latency of the preamble P_{DSP}, which is 0.444s. From Table 1, we conclude that execution scaling factor $m = 11$ still meets the end-to-end latency and memory constraints. The impact of execution scaling factor $m = 11$ on the number of instruction and data cache misses is 6.4 and 3.4, respectively. The impact on the total number of cache misses (instruction plus data) is a factor 4.2.

For the experiments in this paper, we adapted the size of the instruction and data cache to show the impact of execution scaling on the number of cache misses. In general, if cache sizes are fixed, we can change the actor granularity and allow execution scaling to optimise for cache misses.

m	Capacity [KByte]	Latency [s]	m	Capacity [KByte]	Latency [s]
1	40	0.507	7	146	0.772
2	54	0.542	8	165	0.818
3	72	0.588	9	184	0.864
4	94	0.645	10	202	0.910
5	108	0.680	11	226	0.968
6	127	0.726	12	241	1.003

Table 1. Total FIFO buffers capacity and end-to-end latency for different scaling factors m

6. Conclusion

We proposed a novel cache aware mapping technique that reduces the number of instruction and data cache misses for streaming applications in a multiprocessor system. It is shown that executing actors multiple times in a loop, is effective if the individual actors fit in the instruction and data cache, and the set of actors executed on a processor do not fit simultaneously. We have introduced a CSDF model for an application mapped onto a multiprocessor and a specific execution scaling factor. With this model we derived the maximum number of successive actor executions, by making use of traditional dataflow analysis techniques. For our industrial case study, which is a Digital Radio Mondiale receiver, we reduced the number of cache misses by a factor 4.2. The reduction of the number of cache misses and the reduction of contention at the external memory, will improve the overall system performance.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, Feb 1996.
[2] D. Culler, J. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach.* Morgan Kaufmann Publishers, Inc., 1999.
[3] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Proc. Int'l Symposium on System Synthesis (ISSS)*, 2001.
[4] J. Gee, M. Hill, D. Pnevmatikatos, and A. Smith. Cache performance of the spec92 benchmark suite. *IEEE Micro*, 13(4):17–27, Jul/Aug 1993.
[5] A. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency minimization for synchronous data flow graphs. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.
[6] J. Hennessy and D. Patterson. *Computer Architecture A quantitative Approach.* Morgan Kaufmann Publishers, 2003.
[7] S. Kohli. Cache aware scheduling for synchronous dataflow programs. Master's thesis, University of California, Berkeley, CA, 2004.
[8] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Decoupling of computation and communication with a communication assist. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, 2007.
[9] O. Moreira and M. Bekooij. Analysis of self-timed schedules for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.
[10] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.
[11] S. Rathnam and G. Slavenburg. An architectural overview of the programmable multimedia processor, tm-1. In *Proc. Int'l Computer Conf. (COMPCON)*, 1996.
[12] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proc. Int'l Conf. on Application-Specific Array Processors*, 1993.
[13] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proc. Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
[14] J. Stott. Digital radio mondiale: key technical features. *Electronics and communication engineering journal*, 14(1):414, Feb 2002.
[15] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proc. Design Automation Conference (DAC)*, 2006.
[16] M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proc. Design Automation Conference (DAC)*, 2007.
[17] www.systemc.org. Systemc community.