Multithreaded Code from Synchronous Programs: Generating Software Pipelines for OpenMP

Daniel Baudisch, Jens Brandt, and Klaus Schneider
Embedded Systems Group
Department of Computer Science, University of Kaiserslautern
http://es.cs.uni-kl.de

Abstract

In this paper, we describe an automatic synthesis procedure that translates synchronous programs to software pipelines, i.e. to multithreaded software systems whose sequential threads form single stages of a pipeline which are connected by FIFO buffers. The single-loop form that is required for pipelining is thereby already given by a pre-processing step that translates the synchronous programs to synchronous guarded actions. These synchronous guarded actions are analyzed in terms of their data-dependencies to define legal partitions into pipeline stages. Given such a legal partitioning into pipeline stages, the presented synthesis procedure automatically identifies potential pipeline conflicts and implements code for forwarding (if possible) while stalling is implicitly given by the FIFO buffers. Finally, the sequential threads for the conflict-free pipeline stages are implemented in OpenMP-based C-code. We demonstrate the usefulness of our approach by some preliminary experimental results.

1 Introduction

Multicore processors have already replaced single-core processors in desktop computers, and they are more and more frequently used in embedded systems. Hence, the software code generators used in model-based design for the development of embedded systems have to be modified accordingly so that the increased performance offered by multicore processors can be effectively utilized.

In general, there are many ways to write multithreaded code. Most of them require that the programmer already implements his/her system description with the threads that finally run on the multithreaded hardware. In this paper, we follow a different approach and consider the partitioning of system descriptions into threads that are not directly given by the original description. Instead, we use a synchronous language for the system description and show how threads can be automatically generated from such a description to form a pipeline whose stages are implemented by the identified software threads. The reasons for our particular choices are explained in the remaining paragraphs of the introduction.

Pipelining is a simple principle for parallel computation that dates back to the late 1950s [15, 20]. It can be applied to all kinds of processes that repeatedly apply a sequence of actions $\alpha_1; \ldots; \alpha_p$ to an incoming stream of objects. Instead of processing single actions for each object one after the other, a pipelined system processes $p$ objects $o_{t+p}, \ldots, o_{t+1}$ at every point of time $t$ in parallel and applies thereby the actions $\alpha_1; \ldots; \alpha_p$ to these objects in parallel (i.e., action $\alpha_{i+1}$ is applied to object $o_{t+p-i}$ at time $t$). Pipelining is generally used to increase the throughput of the system: Using $p$ pipeline stages, a theoretical speed-up by a factor $p$ can be obtained [16] without increasing the number of actors that perform the actions $\alpha_i$. Essentially all microprocessors are nowadays implemented with pipelines to speed-up the processing of their instruction streams. However, pipelining is a much more general parallel
processing technique and can therefore also be used to create pipelines of software threads. A very good survey on the architecture and analysis of pipelines (which is absolutely worth reading for its historical viewpoint) can be found in [20].

To apply pipelining to a system description, we assume that the considered description consists of a single loop whose loop body \( \alpha_1; \ldots; \alpha_p \) can be pipelined as outlined above [20]. Clearly, this is a strong requirement that can, however, be always achieved by appropriate program transformations. As we will explain in the next section, our compiler for synchronous languages is already able to perform this required program transformation which is one of our motivations for choosing a synchronous language as starting point of our model-based design flow. A further argument for using synchronous languages is that we typically find more concurrency in the programs that allows us to generate more powerful pipelines compared to sequential programs. Finally, the precise notion of time given by synchronous languages allows a formal analysis of the correctness of the generated pipelines.

The generation of software pipelines as presented in this paper should not be confused with software pipelining [17]. The latter is a well-known technique to increase the degree of instruction-level parallelism of loops in sequential programs by unrolling the loops and overlapping a couple of loop bodies (if the data dependencies allow this). This approach is frequently applied by compilers targeting VLIW or superscalar processor architectures [17] — especially for VLIW/EPIC processors [10–12]. The difference between this software pipelining and our approach is that our approach is applied to a level of abstraction much higher than the instruction level considered typically in software pipelining: our pipeline stages are entire threads extracted from a system description. Furthermore, we do not unroll loops to increase the level of concurrency, and instead only partition the given set of synchronous actions into pipeline stages.

Pipelining software in our sense has been only proposed in very specific contexts for multicore processor architectures [9, 21]. For example, Rangan et al. use it to accelerate loops which traverse pointer data structures on VLIW processors. By separating the more or less random memory accesses to these data structures from the actual computation into different threads, pipeline stalls due to caches misses can be significantly reduced.

Moreover, one should not confuse the approach presented here with a related approach we published in [1]: Having constructed a dependency graph of the guarded actions generated by our compiler, the approach shown in [1] splits the graph into vertical slices that form independent threads. Instead, the approach presented in this paper splits the graph into horizontal slices that form threads that implement pipeline stages. Another important difference to [1] is that the threads generated by the horizontal partitioning of the data-dependency graph communicate via FIFO buffers instead of a shared memory which allows us to run them asynchronously in contrast to the threads in [1] (which implement parts of one reaction step).

The approach presented here is therefore closer to Sutherland’s micro-pipelines [29]. As the generated threads run asynchronously to each other, we moreover change the model of computation: we generate delay insensitive [30] threads from a synchronous program.

The rest of the paper is organized as follows: Section 2 briefly introduces synchronous guarded actions, which serve as a starting point for the synthesis procedure of this paper. Section 3 explains how we analyze the data-dependencies of the guarded actions by means of an action-dependency graph (ADG). Given a partitioning of the guarded actions into pipeline stages, we then introduce FIFO buffers in between the pipeline stages so that the stages can run asynchronously to each other. Delayed assignments can lead to pipeline conflicts that have to be resolved accordingly. To this end, our first implementation makes use of pipeline stalling that is implicitly implemented by the FIFO buffers. In Section 4, we show how forwarding can be implemented in our software pipelines to resolve the pipeline conflicts more efficiently. Finally, after showing some experimental results, we conclude with a short summary in Section 5.
2 Synchronous Guarded Actions

Synchronous systems [2, 13] as implemented by synchronous languages like Esterel [3] and Quartz [22, 25] divide their computation into single reactions. Within each reaction, new inputs are synchronously read from all input ports, and new outputs are synchronously generated on all output ports with respect to the current state of the system and the current inputs. Furthermore, the reaction determines the state for the next reaction. It is very important for synchronous languages that variables do not change during the macro step. For this reason, all micro steps are viewed to be executed at the same point of time (as they are executed in the same variable environment). The instantaneous feedback due to immediate assignments to outputs can therefore lead to so-called causality problems [4, 26, 27]. Compilers check the causality of a program at compile time with a fixpoint analysis that essentially correspond to those used for checking the speed-independence of asynchronous circuits via ternary simulation [6]. Besides the causality analysis, compilers for synchronous languages often perform further verification phases to avoid runtime exceptions like out-of-bound overflows or divisions by zero. Moreover, most compilers for synchronous languages also allow the use of formal verification, usually by means of model checking.

The compilation done by our Averest system\(^1\) is split into several stages: The front-end translates a synchronous program into an equivalent set of (synchronous) guarded actions [7, 8, 14, 19] of the form \(\langle \gamma \Rightarrow A \rangle\) (see [5, 22, 25]). The Boolean condition \(\gamma\) is called the guard and \(A\) is called the action of the guarded action, which corresponds to an action of the source language. In our case, these are the assignments of the source language, i.e. the guarded actions have either the form \(\langle \gamma \Rightarrow x = \tau \rangle\) (for an immediate assignment) or \(\langle \gamma \Rightarrow \text{next}(x) = \tau \rangle\) (for a delayed assignment). In each macro step, the guards \(\gamma\) of all actions are checked simultaneously. If a guard \(\gamma\) is true, the right-hand side \(\tau\) of the action is immediately evaluated. Immediate actions \(x = \tau\) assign the computed value immediately to the variable \(x\), while the updates of delayed actions \(\text{next}(x) = \tau\) are deferred to the following macro step. If no action sets the value of a variable in the current step, it is set by the so-called reaction to absence, which either keeps the value of the previous step or resets the value to a default value according to the declaration of the variable. For the sake of simplicity, we only consider variable declarations in the following where the variable’s values are stored if no assignment determines the value of the variables.

Hence, if an immediate assignment \(x = \tau\) is enabled in the current macro step, the current value of \(x\) must equal to the value of \(\tau\). Implementations must therefore make sure that \(x\) is not read before the value of \(\tau\) is evaluated so that one implements the programmer’s view that the assignment was performed in zero time.

Synchronous systems are always deterministic, because there is no choice among activated guarded actions, since all of the enabled actions must be fired. Hence, any system is guaranteed to produce the same outputs for the same inputs. However, forcing conflicting actions to fire simultaneously leads to semantic problems. This is a well-studied problem for synchronous systems and many analysis procedures have been developed to spot and eliminate these problems [23, 24, 27, 28]. In the following section, we assume that a program is causally correct and that for each variable at most one action is active in a macro step.

\[
\begin{align*}
    a \land \neg c & \Rightarrow x = i \\
    c & \Rightarrow x = z \\
    o & = x \\
    x \neq 0 & \Rightarrow \text{next}(z) = x + 1 \\
    i \neq 0 & \Rightarrow a = \text{true}
\end{align*}
\]

Figure 1: Synchronous Guarded Actions

\(^1\)http://www.averest.org
Figure 1 shows a set of synchronous guarded actions, which are used as a running example in the following. Note that the translation of synchronous programs into guarded actions is already the first step towards our generation of software pipelines, since the guarded actions can be viewed as part of a loop body that is repeatedly executed in each reaction step. Hence, our existing compiler front-end already performs the transformation of the system description into the single-loop form required for pipelining.

3 Basic Pipelining

In this section, we will present a first version of our pipelining approach, which will be refined in the following section. Basically, it consists of the following steps: First, we analyze the dependencies between the actions in order to find a partitioning into pipeline stages that respects these dependencies. Second, we create pipeline variables to store intermediate results between stages so that they can run in parallel. Third, we translate the actions of the original system to corresponding actions of the pipelined implementation.

Hence, we first focus on the dependencies between the actions. They can be illustrated by an Action Dependency Graph (ADG), which is a bipartite graph consisting of vertices \( V \) representing variables, vertices \( A \) representing the guarded actions and labeled edges representing the dependencies between the actions and the variables. Thereby, a solid (or dashed) edge from \( \langle \gamma \Rightarrow A \rangle \) to \( x \) denotes that action \( A \) writes \( x \) in the current step (or next step). Similarly, a (solid) edge from \( x \) to \( \langle \gamma \Rightarrow A \rangle \) expresses that \( x \) is read in \( A \), i.e. it appears in the guard \( \gamma \) or in the right-hand side of action \( A \). The set of read and write variables is given by the following definition:

**Definition 1 (Read and Write Dependencies)** Let \( FV(\tau) \) denote the free variables occurring in the expression \( \tau \). Then, the dependencies from actions to variables are defined as follows:

\[
\begin{align*}
\text{rdVars}(\gamma \Rightarrow x = \tau) & := FV(\tau) \cup FV(\gamma) \\
\text{wrVars}(\gamma \Rightarrow x = \tau) & := \{x\} \\
\text{rdVars}(\gamma \Rightarrow \text{next}(x) = \tau) & := FV(\tau) \cup FV(\gamma) \\
\text{wrVars}(\gamma \Rightarrow \text{next}(x) = \tau) & := \{\text{next}(x)\}
\end{align*}
\]

The dependencies from variables to actions are determined as follows:

\[
\begin{align*}
\text{rdActs}(x) & := \{\gamma \Rightarrow A \mid x \in \text{rdVars}(\gamma \Rightarrow A)\} \\
\text{wrActs}(x) & := \{\gamma \Rightarrow A \mid x \in \text{wrVars}(\gamma \Rightarrow A)\}
\end{align*}
\]

Thus, the graph exactly encodes the restrictions for the execution of the guarded actions of a synchronous system. An action can be only executed if all read variables are known. Similarly, a variable is only known if all actions writing it in the current step have been evaluated before.

As already mentioned above, the next step of our pipelining approach is the partitioning of the system into pipeline stages, which serves as the basis for the division into threads. Obviously, the actual choice of partitions has a significant influence on the later performance, and it highly depends on the particular target architecture. In this paper, we do not focus on this question and assume instead that a valid partitioning is given, i.e. a partitioning that respects the action dependencies as defined in the previous paragraphs.

**Definition 2 (Legal Partitioning of an ADG)** A partitioning of an ADG is a mapping from actions to stages, i.e. integers, and vice versa: \( \text{stage}(A) \) is the stage of an action \( A \in A \), and all the actions occurring in stage \( \pi \) are given by \( \text{gacts}(\pi) \). A partitioning is legal iff

\[
\forall A_1, A_2 \in A. (\text{wrVars}(A_1) \cap \text{rdVars}(A_2) \neq \{\}) \rightarrow \text{stage}(A_1) \leq \text{stage}(A_2)
\]
Note that the intersection of $\text{wrVars}(A_1)$ and $\text{rdVars}(A_2)$ is empty according to Definition 1, if $A_1$ is a delayed action for a read variable of $A_2$.

In this section, we further assume that a variable is either written by immediate actions or by delayed actions that are furthermore all within the same stage. Many real-world examples already fulfill this requirement. Otherwise, the introduction of auxiliary variables as intermediate stores can always achieve this requirement. We will not detail this procedure, since this restriction will be removed in Section 4.

The left hand side of Figure 2 shows the ADG of the actions of Figure 1. It is partitioned into three pipeline stages, which can be easily verified to be legal. This ensures that the pipeline created in the following will be free of deadlocks.

Before we introduce pipeline variables, we need to preprocess the delayed actions of our system. In order to guarantee a correct information flow for the basic pipelining, we must ensure that, if a variable is written by delayed actions, then one of them actually fires in each macro step. This can be ensured by explicitly adding the reaction to absence, which transfers the current value to the following step (if no other assignment takes place). Obviously, this action must be triggered if no other action fires, i.e. if the disjunction of all guards of a variable with delayed actions is false. The function $\text{NormalizeBackward}(\mathcal{G})$ (see Figure 3) creates these additional actions for a set of guarded actions $\mathcal{G}$. In our running example, this leads to the creation of the rightmost action for variable $z$ in Stage 3 (see right-hand side of Figure 2).

Since all stages of a pipelined system process data from different macro steps, additional variables are generally needed to store partial results between stages. Therefore, we insert pipeline variables between stages which write and read a given variable. To determine the exact stage boundaries where variables have to be inserted, we recall liveness of classical compiler design and based on this, we define activity of a variable $x$. 

---

**Figure 2: Left: Partitioned ADG, Right: Pipelined ADG with Pipeline Variables**
Definition 3 (Access and Activity) For all stages $\pi \in \{1, \ldots, N\}$ and for all variables $x$, the predicates $\text{read}(x, \pi)$ and $\text{write}(x, \pi)$ denote whether $x$ is read or written in stage $\pi$, respectively.

$$\text{read}(x, \pi) = \exists G. G \in \text{gacts}(\pi) \land x \in \text{rdVars}(G) \quad \text{write}(x, \pi) = \exists G. G \in \text{gacts}(\pi) \land x \in \text{wrVars}(G)$$

For the definition of activity, we assume $\text{write}(x, 0)$ for each input variable $x$ and $\text{read}(x, N)$ for each output variable $x$. Then, a variable $x$ is said to be active in stage $\pi$, written $\text{active}(x, \pi)$, if variable $x$ is read or written in an earlier stage and is read in a later stage - or formally:

$$\text{active}(x, \pi) = (\exists i < \pi \text{write}(x, i) \land \exists j \geq \pi \text{read}(x, j)) \lor (\exists i < \pi \text{read}(x, i) \land \exists j \geq \pi \text{read}(x, j))$$

With the help of the activity definition, we can identify the pipeline stages for which we must add a representative for variable $x$. While its first part reflects the traditional definition of liveness, the second part is needed for variables that are only written by delayed actions (e.g. variable $z$ in Figure 2). For these variables, there is no write action that precedes the first read action (in spatial dimension).

If $\text{active}(x, \pi)$ holds, a pipeline variable for $x$ is needed between stage $\pi$ and its preceding stage. To distinguish between the different pipeline variables of $x$, we add a superscript $\pi$, where $x^\pi$ represents the pipeline variable for $x$ between stage $\pi$ and its succeeding stage $\pi + 1$. Pipeline variables are also called incarnations. Among the incarnations $x^i$, we select a canonical one for each variable $x$. The canonical position marks the point in the pipeline where a variable $x$ must have become stable: previous incarnations may still contain invalid values, while successive ones must comply with the canonical incarnation - formally: $\text{canon}(x) = \min\{\pi \mid \text{read}(x, \pi)\}$.

Definition 3 guarantees that all input variables are contained in the first set of pipeline variables, and all output variables are contained in the last set. Thus, externally visible variables are synchronized, i.e. all input variables of a macro-step are read simultaneously, and all output variables of a macro-step are written simultaneously.

Since all original variables have now been replaced by the pipeline variables, the guarded actions must be rewritten to refer to them. Apparently, all immediate actions of the original system, which are put in stage $\pi$ read variables with superscript $\pi - 1$ and write variables with superscript $\pi$. Delayed actions in stage $\pi$ also read variables with superscript $\pi - 1$ and write to the canonical incarnation of this variable. Hence, we rewrite the actions as given in the function BasicTransform shown in Figure 3. Thereby, let $[\gamma]^\pi$ be the operations that relabel the variables $x \in \text{FV}(\gamma)$ with their incarnations $x^\pi$.

Finally, we have to add the transport of the pipeline variables, which corresponds to the reaction to absence of a synchronous system: if the current stage does not contain an action which actively writes the variable, the value of the previous pipeline variable must be carried over. If no such variable exists, an arbitrary value (e.g. the default value Default$(x)$) can be used. These actions are determined by the function CreateTransport given in Figure 3.

The right-hand side of Figure 2 shows the transformed set of guarded actions including the transport actions for our running example. The five original actions are rewritten so that they refer to the incarnations, and the remaining actions are due to the transfer of pipeline variables.

This transformed system is now taken as the basis for an OpenMP implementation. OpenMP is an API based on compiler directives for shared-memory parallel programming in C/C++ and Fortran on multiple platforms. It uses the fork-join model to execute programs in parallel, i.e. in addition to executing a sequence of usual statements, each thread can fork into several child threads (often called team), which all execute different tasks. All threads have an implicit barrier at their end, which forces them to join before the execution of their parent thread resumes. OpenMP provides several compiler directives for the creation of threads, and it helps developers to control and synchronize access to global variables.

Our translation only makes use of some of the directives. First, to create parallel code, we use the parallel sections directive, which marks a fork in the current thread and thereby creates a new team. To describe each of its members, we use the section directive. Hence, we create sequential C code for
function NormalizeBackward(G)
    \( G' := \{ \} \)
    forall \( x \in G \)
        \( G_x := \text{wrActs}(\text{next}(x)) \)
        \( \sigma := \neg \bigvee_{(\gamma \Rightarrow A) \in G_x} \gamma \)
        \( G' := G' \cup \{ \sigma \Rightarrow \text{next}(x) = \text{Default}(x) \} \)
    return \( G \)

function BasicTransform(G)
    \( G' := \{ \} \)
    forall \( G \in G \)
        \( \pi := \text{stage}(G) \)
        case \( G \):
            \( \gamma \Rightarrow x = \tau \):
                \( G' := G' \cup \{ \gamma \Rightarrow [x]^\pi = [\tau]^\pi \} \)
            \( \gamma \Rightarrow \text{next}(x) = \tau \):
                \( G' := G' \cup \{ \gamma \Rightarrow [x]^\pi \Rightarrow [x]^\text{canon}(x) = [\tau]^\pi \} \)
    return \( G' \)

function CreateTransport(G)

for \( \pi = 1, \ldots, N \)
    forall \( x^\pi \in V \)
        \( G_{x^\pi} := \text{wrActs}(x^\pi) \)
        \( \sigma := \neg \bigvee_{(\gamma \Rightarrow A) \in G_{x^\pi}} \gamma \)
        if \( x^{\pi-1} \in V \)
            \( G := G \cup \{ \sigma \Rightarrow x^\pi = x^{\pi-1} \} \)
        else
            \( G := G \cup \{ \sigma \Rightarrow x^\pi = \text{Default}(x^\pi) \} \)
    return \( G \)

function BasicPipeline(G)
    \( G := \text{NormalizeBackward}(G) \)
    \( G := \text{BasicTransform}(G) \)
    \( G := \text{CreateTransport}(G) \)
    return \( G \)

Figure 3: Basic Pipelining Functions

each pipeline stage, put it into an infinite loop, which is again put into an own section. Second, inputs, outputs and variables that are written and read by successive pipeline stages are declared as shared by the identical OpenMP directive. Instead of using simple variables, which can only store a single value, we implement the pipeline variables by FIFO buffers. Thereby, the individual pipeline stages do not have to synchronize at the end of each step, which significantly speeds up the execution. Furthermore, this design choice relieves us from adding an explicit control logic, which is usually needed in hardware implementations to control bubbles stalls. For a correct execution, the synthesis does not rely on a specific size of the buffers: in principle, it can be only one element (which, in general, will not be optimal w. r. t. performance).

A lightweight synchronization on the FIFO can be realized as follows: As the previous construction guarantees that a pipeline variable is only written by a single stage (i.e. single thread) and only read by another one, required synchronization can be implemented by Lamport’s clocks [9, 18], which involve little overhead.

The synthesized system still produces for the same sequence of inputs the same sequence of outputs, although we have desynchronized the original version. Correctness is due to our construction that puts exactly one value for each pipeline variable in the corresponding FIFO buffer. Thereby, the buffer transports the values for the corresponding pipeline variables for all synchronous threads. Furthermore, for each buffer there is a single designated stage, which is responsible for filling the buffer. This ensures that the elements in this buffer are in the correct order, and races between concurrently produced values are impossible.

4 Optimizations

An apparent drawback of the pipelining approach presented in the previous section consists of potentially long stalling periods of pipeline stages due to the condition that at most one pipeline stage can write to a buffer. For instance, consider our running example and assume that there would be another delayed action writing \( z \) in Stage 2. Since different stages execute different macro steps, assignments to a variable could
be executed out of order and race conditions may appear, which may destroy correctness. Hence, the approach of the previous section would pass the result computed in Stage 2 to the Stage 3, which then would transfer it to the correct buffer - but one or more steps later. In general, all delayed actions would have to transport their result to succeeding stages, and only the last one writes its result to the canonical incarnation.

Obviously, this is not a good solution because results may be kept back and block other stages from execution. These unnecessary delays can be avoided if several stages have the possibility to write to a buffer, in particular to the canonical incarnation, thereby mimicking forwarding of classical processor design. The core of our optimized approach consists exactly of this extension, which allows any action to write to each buffer. As a side effect, we can also neglect the restriction given at the beginning of Section 3, which limited this random access.

This relaxation leads to several small modifications of our basic approach: First, we redirect all actions to the canonical incarnation of the corresponding variable, leading to a slightly modified algorithm for creation of the pipeline (see modified OptimizedTransform in Figure 4). Thereby, some pipeline variables of the basic approach are not needed any more. The value of $x$ has only to be carried over to stages that read $x$. Hence, we redefine activity as follows: active($x$, $\pi$) = ($\exists_i \leq \pi \ read(x, i)$ $\land$ $\exists_j \geq \pi \ read(x, j)$).

However, the price we have to pay is to add a mechanism that retains the correct temporal ordering of values under the relaxed context. This is provided by merge components, which are inserted before the canonical incarnation of each variable $x$ in stage $\pi$. They reorder values coming from all stages that write to $x$ in the original ADG. To select the right value, they make use of logical timestamps, which are attached to output values by all writing stages: each time an immediate action (delayed action) for variable $x$ is fired, the responsible action sets the timestamp of the output value to the same value (next value) as the one of the input set that is currently processed. The merger can then select the requested value and forward it to the canonical incarnation. Due to the absence of write conflicts in our synchronous system model, each macro step contains at most one assignment to a variable, and the absence reactions ensure that each macro step contains at least one assignment to each variable. Thereby, it is guaranteed that the merger eventually gets exactly one value for the requested step. In addition, the merger is also able to react to absence of variable values, resulting in an optimized creation of pipelines as illustrated in OptimizedPipeline in Figure 4. The point of time when an absence reaction for an input set $I$ has to be fired can be determined by inserting two actions. These actions notify the merger that no immediate or delayed actions are left that may modify $x(I)$, no matter if $x(I)$ has already been set by an action. Hence,
the merger has to discard these signals, if the corresponding input set has already been processed. In case of a request of a value that is not available, the merger executes the reaction to absence iff both signals for this variable are available.

The right hand side of Figure 4 illustrates how the merger processes data. As can be seen, the merger provides one or two queues for each stage that contains write actions for the corresponding variable and one queue for each absence notification. The number of queues for each stage depends on whether a variable is either written immediate or delayed (one queue) or it is written immediate and delayed (two queues). Because, each stage calculates its outputs in order, the values in each queue are also temporally ordered. Hence, the merger only needs to search at the front of each queue for the currently requested value.

5 Preliminary Results and Summary

To evaluate the feasibility of our approach, we applied the optimized pipelining procedure to a parallel mergesort procedure and then executed the OpenMP-based implementation on two different multicore systems: a Pentium D (two cores) and a Dual Xeon Quad Core (eight cores). Mergesort splits the task of sorting an array of size $n$ into $\log_2(n)$ sorting stages, where each stage consists of $n$ insert-operations. Hence, the complexity of mergesort is $n \cdot \log_2(n)$, where $n$ is the size of the array. To utilize a system with $l$ cores, the theoretical minimum size of the arrays has to be $2^l$. For our benchmark, we compare two different synthesized versions of the sorting program: a single-threaded version and a multithreaded version, which uses two threads for the Pentium D and eight threads for the Dual Xeon Quad Core. For each, we used the same 100 randomized arrays.

Figure 5 shows the results of the benchmarks. On both platforms, the multithreaded implementations take more time for small input sizes as the singlethreaded version, but soon reaches break-even. As expected, the eight-core system needs larger arrays to reach this point, which is due to the higher synchronization costs of its eight partitions. Nevertheless, a speed-up of 5 could be reached in our first experiments.

To conclude, this paper shows an approach to derive multithreaded implementations of synchronous programs. Thereby, the original system does not need to be divided into threads, but they are automatically generated by cutting the original system into pipeline stages. It is based on pipelining these programs before turning them into OpenMP-based C-Code. By connecting all parts of the implementation by FIFO buffers, the execution of the stages can be desynchronized. First experimental results show the feasibility of our approach.
References


