Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set

Lars Bauer, Muhammad Shafique, Stephanie Kreutz, Jörg Henkel
University of Karlsruhe, CES − Chair for Embedded Systems, Karlsruhe, Germany
{lars.bauer, shafique, henkel} @ informatik.uni-karlsruhe.de

Abstract

One of the upcoming challenges in embedded processing is to incorporate an increasing amount of adaptivity in order to respond to the multifarious constraints induced by today's embedded systems that feature complex and diverse application behaviors.

We present a novel concept (evaluated with a hardware prototype) that moves traditional design-time jobs to run time in order to increase efficiency (in this paper we focus on performance). Adaptivity is achieved dynamically through what we call Special Instructions (SIs) which may change during run time according to non-predictable application behavior. The new contribution of this paper is the principal component that actually makes the entire embedded processor work efficiently, namely the "Special Instruction Scheduler". It determines during run time 'when' and 'how' Special Instructions are composed and executed.

We achieve a 2.38x performance increase over a reconfigurable processor system with dynamic instruction set (Molen [19]). Our whole platform consists of a toolchain including estimation and simulation tools plus a running hardware prototype. Throughout this paper, we discuss the functionality by means of an H.264 video encoder in detail even though the concept is not limited to this application.

1. Introduction and motivation

Embedded processors are key components for rapidly growing application fields ranging from automotive to personal mobile communication/entertainment etc. *ASIPs* (Application Specific Instruction Set Processors) have proven to be very efficient in terms of performance per chip area, performance per power consumption etc. However, today's landscape of embedded applications is rapidly changing as we see more complex functionalities, which makes it increasingly difficult to estimate a system's behavior sufficiently accurate at design time.

In fact, after extensive exploration of complex real world embedded applications, we have found that it is hard or even impossible to predict the performance and other design criteria accurately during design time. Consequently, the more critical design decisions are fixed during design time, the less flexible an embedded processor can react to non-predictable application behaviors. Hence, the embedded processor performs the lesser efficient, the more complex the application is.

We have analyzed state-of-the-art embedded processors and encountered a problem of inefficient utilization of hardware resources. We will illustrate this problem by means of an ITU-T H.264 video encoder [8]. During the execution, the processing flow migrates from one computational hot spot to another, i.e. from *"Motion Estimation"* (ME) to *"Encoding Engine"* (EE) to *"Loop Filter"* (LF) and back to ME. This movement demands a switch from one set of custom instructions to another, since the requirements within these hot spots are different.

State-of-the-art *ASIPs* typically provide dedicated hardware (e.g. in form of Special Instructions (SIs)) to efficiently address hot spots. However, when many diverse hot spots are addressed it means that many SIs need to be provided. Hence, a significant hardware overhead is the result. According our studies with truly large and inherently diverse applications, the necessary overhead can easily grow twice the size of the original processor core. Since often only one hot spot is executed at a certain time, the major hardware resources reserved for other hot spots are idling. This indicates an inefficiency that is an implication of the extensible processor paradigm. In the example of Figure 1a) it is visible that during the execution of ME, EE and LF are idling.

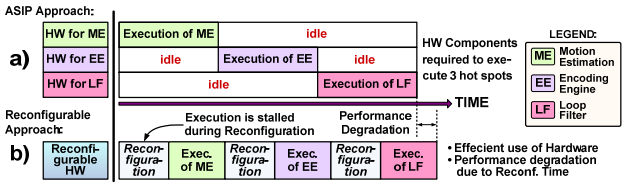


Figure 1: Efficiency problems of typical embedded architectures targeting applications with multiple hot spots

Reconfigurable computing may address this problem of inefficiently utilized resources through reconfigurable hardware ([5], [17], and [18]). All SIs executed in a certain period may utilize the ENTIRE hardware resources dedicated to ALL SIs (since the currently non-executing SIs may then vacate their not needed hardware resources). Figure 1b) shows that the hardware resource for ME is loaded first. During this time, the ME execution is stalled. When the reconfiguration is finished, ME starts execution in hardware[1]. Afterwards, the hardware for EE is re-loaded. The reconfiguration latencies contribute to overall performance degradation. A simple solution to 'hide' the reconfiguration latencies is to process an SI using the instruction set of the general purpose base processor until the required hardware is reconfigured. However, even that concept is inefficient, as we will see.

In our approach, we decompose an SI into connected data path modules to solve the above problem. After the re-loading of one data path is completed, SIs may use it for execution. Even though an SI is not finally composed (i.e. not yet fully efficient), it may already be used for execution. After the re-loading of additional data paths, this SI may be gradually upgraded. In this way, a data path is functional immediately, i.e. on an *"as-soon-as-available basis"*. An SI may be executed utilizing different combinations of these data paths (but still maintain its functionality) depending on what is available at a certain point in time. The idea is to provide access to different data paths as independent processing units such that SIs can switch between base processor instructions and (reconfigurable) data paths. This facilitates the gradual upgrading of an SI, as we will see in section 3.

We have analyzed the ME processing using SIs with and without upgrade feature. Figure 2 shows the in-depth view of the reconfiguration and execution of two SIs (i.e. SAD and SATD) used for ME processing. The X-axis presents the execution time, while the Y-axis shows how often the SIs are executed per 100K cycles execution time. For the dashed line (without SI upgrade) until around 160K cycles both SIs are executed using the base processor instruction set. Although at this point the SAD hardware is reconfigured, the ME loop is not fully expedited due to the still slow execution of SATD. After completing the reconfiguration of SATD (at around 700K cycles), the total number of SI executions per period increases significantly, thus showing a large speed up for the ME process. The continuous line in Figure 2 shows the processing of ME using stepwise SI upgrades. After 300K cycles the ME process is rapidly expedited, as both SIs are available in a hardware implementation (although not at full performance yet). Therefore, the version with SI upgrades finishes the executions earlier than the one without SI upgrade.

[1] but therefore in a potentially faster implementation (compared to ASIP), as all available hardware may be spend for acceleration

978-3-9810801-3-1/DATE08 © 2008 EDAA

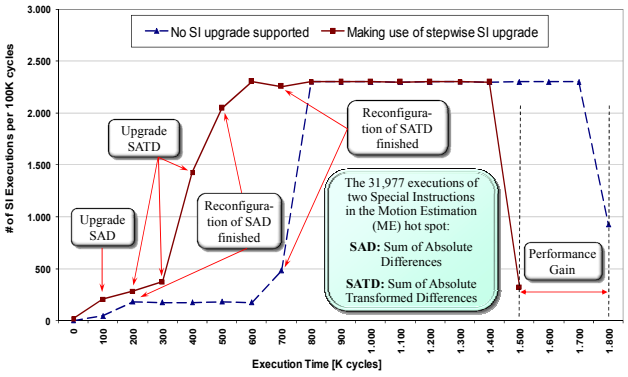


Figure 2: Comparing the SI executions with and without upgrade support for performing the Motion Estimation

Although the modular composition of SIs provides a gradual upgrade capability, still there is one major challenge in this approach to obtain the full benefit: the problem of dynamically determining the loading sequence of the data paths especially when multiple SIs are needed in order to accelerate a large hot spot. Here arises the need of an **efficient run-time scheduler**[2].

Abridging the whole motivation, in order to run an embedded processor with dynamic instruction set most efficiently, the "SI Scheduler" (determining the SI upgrading sequence) is key.

Our novel contributions are:

- Modular Special Instructions (SI) that are gradually composed out of data paths during run time. Thus, an SI can be executed with a mixture of dynamically loaded data paths in conjunction with the base processor instructions to increase the efficiency compared to state-of-the-art extensible embedded processors.
- An enabling run-time "Special Instruction Scheduler" that provides a foundation for the above concept by determining an advantageous (e.g. for performance) loading sequences of data paths, using online monitoring information.

We achieve performance improvements of 2.38x for an implemented H.264 video encoder compared to a state-of-the-art reconfigurable processing systems (like Molen [19]). The concept is by no means limited to a video encoder application. In fact, it is particularly beneficial in all cases where run-time situations occur that cannot be well predicted during design time (like varying workloads, constrains etc.), as the schedule has to reflect these changing situations.

The rest of the **paper is organized as follows**: In Section 2, we present the related work. A system overview including our SI composition and the run-time architecture are presented in Section 3. Section 4 shows the main part of this paper namely the Special Instruction Scheduler that enables run-time adaptation. In Section 5, we discuss the results and we conclude in Section 6.

2. Related work

Since our work partly relates to extensible embedded processors and partly to reconfigurable computing, we review the most prominent work from both areas as far as it is relevant to our approach. Commercial tool suites and processor cores like Tensilica [7], ARC [1], Target [4], CoWare/LisaTek [3], ASIP Meister [2] etc. are nowadays available for designing *ASIPs*. A general overview of the benefits and challenges of *ASIPs* is given in [9]. A major focus in that field is spent in automatically detecting and generating special instructions (SIs) for application speedup and/or power efficiency etc. from the application code [11], [12]. A library of reusable functions is used in [13], whereas in [14], [15] the authors describe methods to generate SIs from matching profiling patterns. [10] concentrates on *ASIP* design space exploration with tool-supported connection to reconfigurable hardware.

An overview for reconfigurable computing can be found in [16]. Besides the approach of targeting full computational tasks in reconfigurable hardware [17], the research for CPU-attached reconfigurable systems mainly focused on design-time predefined reconfiguration decisions. This is not suitable when computational requirements/constraints change during run time and are unpredictable during design time. The Molen Processor couples reconfigurable hardware to a base processor via a dual-port register file and an arbiter for shared memory [19]. The run-time reconfiguration is explicitly predetermined by additional instructions. An overview for reconfigurable computing with a rather tightly coupled interface to the CPU is given in [20]. The OneChip98 project [21] uses a Reconfigurable Functional Unit (RFU) that is coupled to the host processor and that obtains its speedup mainly from streaming applications. The Warp Processor [22] proposes to generate custom logic through micro-CAD on-chip tools at run-time, which incurs a non-negligible waiting time due to hardware synthesis. Therefore, this approach may mainly be beneficial, if the requirements seldom change and thus the on-chip synthesis is only required from time to time.

3. System overview and basic idea

All state-of-the-art architectures are using mostly *one* implementation for each Special Instruction (SI). That limits the potential of run-time adaptivity and encumbers the efficient usage of the already loaded hardware.

Our novel concept enables us to utilize all available hardware resources efficiently on an **as-soon-as-available basis**. That means as soon as data paths are (re-)loaded they may be used to compose the functionality of an SI. Over time, the SIs may then be gradually upgraded to full performance. In what specific composition an SI is available at a certain point is not known at design time since it depends on the context during run time. We now present the hierarchical SI composition as the foundation of our *RISPP* (Rotating Instruction Set Processing Platform) approach [23]:

Atom: An elementary data path; can be re-loaded at run time.

Atom Container (AC): A small reconfigurable region that can be dynamically loaded with one *Atom*.

Molecule: A combination of multiple *Atoms*, implementing an SI. Each SI may have multiple *Molecules* (varying in resource usage & performance) as motivated with Figure 3.

Our concept allows us to trade-off various combinations of compositions for SIs during run time. In other words, it is possible to choose different design points and switch between them when the application is running.

Figure 3 shows the composition of the *Motion Compensation* SI, as we have implemented it for the *Encoding Engine* hot spot in the H.264 application[3] (shown in Figure 1). The internal assembly of the central *PointFilter Atom* is shown besides. The *Atoms* make use of the inherent parallelism and therefore achieve a speedup compared to an execution with the base instructions. We call it the ***Atom*-level parallelism**, which is provided at design time. The SI in Figure 3 can be executed with accelerating hardware even if only one instance of each *Atom*-type (i.e. *BytePack*, *PointFilter*, and *Clip3*) is available. This is done by reusing the single *Atom*-instance for all occurrences of its type. As an extreme, each *Atom*-occurrence in the SI could be implemented by an individual instance of the corresponding *Atom*, thus fully exploiting the so-called ***Molecule*-level parallelism**, which is subject to dynamic adaptation according to changing application/system requirements (see Section 3.1). The slowest implementation of an SI is without any accelerating *Atoms*, only using

[2] this is not a task scheduler, rather it is responsible for arranging the loading sequence of data paths in order to compose SIs

[3] the *Molecules* and *Atoms* in this paper are manually developed for benchmarking our *RISPP* architecture; it is beyond the scope of this paper to automatically determine SIs, as it was done in e.g. [14] or [15]

the base instruction set. It is activated by a synchronous exception (trap) that is automatically triggered if the SI shall be executed, but the required *Atoms* are not yet loaded.

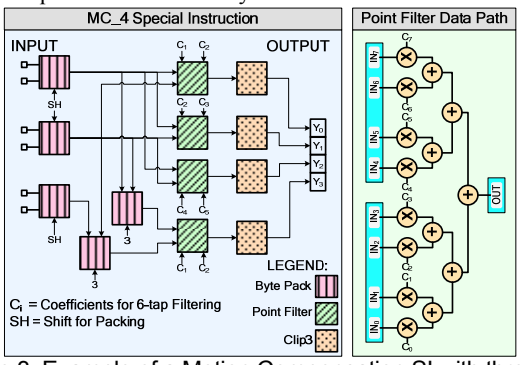


Figure 3: Example of a Motion Compensation SI with three different Atoms and the internal data path of the PointFilter Atom

3.1. Overview of the run-time architecture

Our *Molecule*/*Atom* hierarchy enables us to alter the performance of individual SIs dynamically to suit the application/system requirements. For instance, the encoding-type of a Macro Block (16x16-pixel block) in the H.264 video encoder only depends on the kind of motion in the input video sequence. Depending on the motion type either one or another SI group is more relevant.

We extend a typical CPU pipeline[4] by *Atom Containers* (ACs) and a *Run-Time Manager*, which is controlling the run-time behavior. The ACs are tightly connected to the pipeline. The main tasks of the *Run-Time Manager* are:

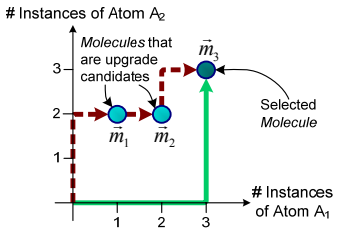
I) Controlling the execution of SIs
II) Observing and adapting to changing/varying constraints
III) Determining *Atom* re-loading decisions

Point I) either forwards the SI input data to the *Atom Containers* or triggers a trap to activate the execution with the base instruction set. Point II) is based on an online monitoring of the SI execution frequencies within a hot spot. After executing the hot spot, this value is compared to the previous expectations to update the expectations for the next execution iteration of this hot spot. The efficiency and light-weighted implementation of this approach was demonstrated in [24].

Finally, point III) has to decide which *Molecules* shall be selected for the upcoming hot spot and in which sequence the *Atoms* shall be loaded. The details of the selection are beyond the scope of this paper. Here, we instead concentrate on the key problem of scheduling the *Atoms* of the selected *Molecules*.

4. Scheduling of Atoms

The job of the scheduling is to determine a loading sequence of the *Atoms* that are needed to implement the selected *Molecules*. This schedule cannot be determined at design time because it highly depends on the selected *Molecules* (and thus on the number of *Atom Containers* and the expected SI execution frequencies). The importance of a good scheduling strategy is shown in a simple example in Figure 4, where the *Molecule* $\vec{m}_3$ was selected to implement a required SI. This SI may also be implemented (with reduced performance) by the *Molecules* $\vec{m}_1$ or $\vec{m}_2$. A good scheduler should exploit the potential for *upgrading* from one *Molecule* to a faster one until the selected *Molecule* is finally composed. Without this incremental *upgrading*, the SI is not available in the accelerating hardware for a noticeable longer time as shown in the table in Figure 4 (the average reconfiguration time for one of our *Atoms* is 874.03 µs, as discussed in Section 5). The problem space of determining an *Atom* loading sequence grows even larger when multiple *Molecules* of different SIs are requested in parallel (which is the typical case) where the scheduler has to decide which SI to upgrade first. A too simplistic decision will give performance away as we will see.



# loaded Atoms	fastest available Molecule	
1	—	—
2	—	—
3	$\vec{m}_1$	—
4	$\vec{m}_2$	—
5	$\vec{m}_2$	$\vec{m}_2$
6	$\vec{m}_3$	$\vec{m}_3$

Figure 4: Different Atom schedules with the corresponding availabilities of the Molecules

4.1. Molecule assembly model and definitions

In this subsection, we explain the hierarchical SI composition of our *RISPP* project on a formal basis (originally published in [23]) to present the *Molecule − Atom* interdependencies and to simplify and clarify expressions for the scheduling problem later on.

We define a data structure $(\mathbb{N}^n, \cup, \cap, \leq)$, where $\mathbb{N}^n$ is the set of all *Molecules* and n is the number of different available *Atom*-types. For convenience we consider $\vec{m}, \vec{o}, \vec{p} \in \mathbb{N}^n$ as *Molecules* with $\vec{m} = (m_1, ..., m_n)$ where m_i describes the desired number of instances of *Atom* i to implement the *Molecule*. The operator $\cup: \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{N}^n$ with $\vec{m} \cup \vec{o} := \vec{p}$; $p_i := \max\{m_i, o_i\}$ describes a *Meta-Molecule* which contains the *Atoms* required to implement both $\vec{m}$ and $\vec{o}$. We name the resulting *Molecule* $\vec{p}$ as a *Meta-Molecule* to distinguish it from the elementary *Molecules* that are dedicated to implement a specific SI. Since the operator $\cup$ is commutative and associative with the neutral element (0, …, 0), therefore $(\mathbb{N}^n, \cup)$ is an Abelian semi-group. The same is true for $(\mathbb{N}^n, \cap)$ with $\cap: \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{N}^n$ being defined as $\vec{m} \cap \vec{o} := \vec{p}$, $p_i := \min\{m_i, o_i\}$ with the neutral element (*maxInt*, …, *maxInt*). The relation $\vec{m} \leq \vec{o}$ is defined to be true iff $\forall i \in [1,n]: m_i \leq o_i$. As '$\leq$' is reflexive, anti-symmetric, and transitive, $(\mathbb{N}^n, \leq)$ is a partially ordered set. For a set of *Molecules* $M \subset \mathbb{N}^n$, the supremum is defined as $\sup M := \bigcup_{\vec{m} \in M} \vec{m}$. The supremum from M is a *Meta-Molecule* with the meaning of declaring all *Atoms* that are needed to implement any of the *Molecules* in M, i.e. $\forall \vec{m} \in M: \vec{m} \leq \sup M$. The infimum is correspondingly defined as $\inf M := \bigcap_{\vec{m} \in M} \vec{m}$ with the meaning of containing those *Atoms* that are collectively needed for all *Molecules* of M. As any subset $\emptyset \neq M \subset \mathbb{N}^n$ has a well-defined supremum and infimum, $(\mathbb{N}^n, \leq)$ is a complete lattice. Given these definitions, we can now combine multiple *Molecules* that are chosen to implement different SIs. To calculate the cost for constituting *Molecules* in hardware, we define the following two functions: The determinant of a *Molecule* is defined as $|\vec{m}| := \sum_{i \in [1,n]} m_i$, i.e. the total number of *Atoms* that are required to implement $\vec{m}$. To consider already configured *Atoms* we define the function

$$\triangleright: \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{N}^n;\ \vec{m} \triangleright \vec{o} := \vec{p};\ p_i := \begin{cases} o_i - m_i, & \text{if } o_i - m_i \geq 0 \\ 0, & \text{else} \end{cases}.$$

The created *Meta-Molecule* $\vec{p}$ contains the minimum set of *Atoms* that additionally have to be offered to implement $\vec{o}$, assuming that the *Atoms* in $\vec{m}$ are already available.

4.2. Defining the Atom Scheduling problem

The input to the scheduling is a set $M = \{\vec{m}_i\}$ of all *Molecules* $\vec{m}_i \in \mathbb{N}^n$ that were selected for implementation. $\sup(M)$ is the *Meta-Molecule* that contains all *Atoms* that are needed to implement the selected *Molecules*. We name the number of instances for the i[th] *Atom* of $\sup(M)$ "x_i", i.e. $\sup(M) = (x_1, x_2, ..., x_n)$. We define $NA := |\sup(M)| = \sum x_i$ as the number of *Atoms* that are needed to implement all selected *Molecules*. The previously executed *Molecule* selection guarantees that these *Atoms* fit to the available *Atom Containers* (ACs), i.e. $NA \leq \#ACs$. We define so called

[4] for evaluation we are working with a DLX (MIPS) and a Leon2 (SPARC V8) based prototype, but we are not limited to these

Unit-Molecules (UM) to represent our elementary *Atoms*: $\vec{u}_1 := (1,0,...,0)$, $\vec{u}_2 := (0,1,0,...,0)$, …, $\vec{u}_n := (0,...,0,1)$. With these definitions, we can now define a loading sequence for *Atoms* as a scheduling function.

$$\mathrm{SF}: [1,k] \rightarrow UM := \{\vec{u}_1,...,\vec{u}_n\} \quad (1)$$

The interval $[1,k]$ hereby represents k consecutive moments where in the moment $j \in [1,k]$ the loading of the *Unit-Molecule* (and thus the single *Atom*) in SF(j) is started. The scheduling function for the better schedule in Figure 4 is $\mathrm{SF}(1)=\vec{u}_2$, $\mathrm{SF}(2)=\vec{u}_2$, $\mathrm{SF}(3)=\vec{u}_1$, $\mathrm{SF}(4)=\vec{u}_1$, $\mathrm{SF}(5)=\vec{u}_2$, $\mathrm{SF}(6)=\vec{u}_1$.

To make sure that exactly those *Atoms* are loaded which are needed to implement the requested *Molecules* we define an additional condition that restricts the general function of (1).

$$\forall i \in [1,n] : \left|\{j \mid \mathrm{SF}(j) = \vec{u}_i\}\right| = x_i \quad (2)$$

The condition (2) ascertains, that SF considers each $\vec{u}_i$ in the correct multiplicity x_i, which guarantees, that $\bigcup_{\forall i \in [1,n]} \mathrm{SF}(i) = \sup(M)$. The function (1) with the condition (2) describes a *valid* schedule. We now have to discuss the properties of a *good* schedule. The main goal is to reduce the number of cycles that are required for executing the current hot spot. Therefore, it is essential to exploit the architectural feature of stepwise upgrading from slower to faster *Molecules* until all selected *Molecules* are available. An *optimal* schedule requires a precise future knowledge (i.e. which SI will be executed when) to determine the schedule that leads to the fastest execution. For a realistic scheduler implementation, we have to restrict on less exhaustive future knowledge. Due to our online monitoring (see Section 3.1) we have an estimate which SI is more *important* (in terms of expected executions) than another one. In summary, we obtain the expected SI executions as additional input.

4.3. Strategies for considering Molecule upgrades

First, we reduce the problem of scheduling *Atoms* to the problem of scheduling *Molecules*. The reason is that major performance changes occur exactly then when, upgrading from an available *Molecule* to a faster one. This strategy not only reduces the scheduling complexity (each scheduled *Molecule* requires at least one additional *Atom*) but it also allows for a clear expression which SI shall be upgraded next. When $\vec{a}$ denotes the already available/scheduled *Atoms* and $\vec{m}$ the *Molecule* that shall be scheduled next, then the additionally required *Atoms* $\vec{a} \triangleright \vec{m}$ are defined to be the next output of the scheduling function (1). The condition from (2) is implicitly assured by the strategy of scheduling the upgrade steps for the selected *Molecules*. But to make use of stepwise upgrading from slower to faster *Molecules* we first need to determine all *smaller* ($\vec{o} \le \vec{m}$) *Molecules* M' that may implement the same SIs as the selected *Molecules* M.

$$M' = \bigcup_{\vec{m} \in M} \{\vec{o} : \vec{o} \le \vec{m} \wedge \vec{o}.getSI() = \vec{m}.getSI()\} \quad (3)$$

These *Molecule* candidates M' are all possible intermediate steps that might be considered on a schedule up to $\sup(M)$. However, some further optimizations have to be considered. On the one hand, a *Molecule* candidate might be already available although it was not explicitly scheduled. This depends on the *Atoms* that were initially available in the *Atom Containers* and the *Atoms* of those *Molecules* that are already scheduled. On the other hand, a currently unavailable *Molecule* is not necessarily faster than the currently fastest available/scheduled *Molecule* for the same SI. For example Figure 4 may contain a third upgrade candidate $\vec{m}_4 = (1,3)$ with a worse latency than $\vec{m}_2 = (2,2)$ (the latency denotes the number of cycles that are required for a single execution of the corresponding *Molecule*). After $\vec{m}_2$ is composed by the schedule with the dashed line (Figure 4), $\vec{m}_4$ is still unavailable and it does not offer a latency improvement. Nevertheless, $\vec{m}_4$ may be beneficial depending on the initially available *Atoms* $\vec{a}$ if $\left|(\vec{a} \triangleright \vec{m}_4)\right| \le \left|(\vec{a} \triangleright \vec{m}_2)\right|$ (e.g. for $\vec{a} = (0,3)$). Therefore, we cannot assume that *Molecules* like $\vec{m}_4$ are removed at compile time. We instead clean the list of *Molecule* candidates from M' to M'' for the currently available or scheduled *Atoms* $\vec{a}$ before we schedule the next *Molecule*.

$$M'' = \left\{ \vec{m} \in M' : \left(\begin{array}{c} |\vec{a} \triangleright \vec{m}| > 0 \wedge \vec{m}.getLatency() < \\ \vec{m}.getSI().getFastestAvailable- \\ Molecule(\vec{a}).getLatency() \end{array} \right) \right\} \quad (4)$$

4.4. Determining the Molecule loading sequence

As the possibility to upgrade from one *Molecule* to a faster one can lead to a significant improved execution time (as motivated through Figure 4), our first proposed scheduling method mainly concentrates on exploiting this feature. In general, multiple SIs are required to accelerate one hot spot. "*First Select First Reconfigure*" (FSFR) concentrates on first upgrading the most *important* SI (in terms of expected SI executions and potential performance improvement due to the selected *Molecule*) until it reaches the selected *Molecule*, before starting the second SI. A simple example for an FSFR schedule is shown in Figure 5. To ease the explanation, we restrict in the discussion to *Molecules* that only use two different kinds of *Atoms*, i.e. (A_1, A_2). The two dark-filled *Molecules* (circle for SI_1 and square for SI_2) are the selected *Molecules* for addressing the current hot spot. The lighter filled *Molecules* are intermediate upgrade possibilities for the two SIs.

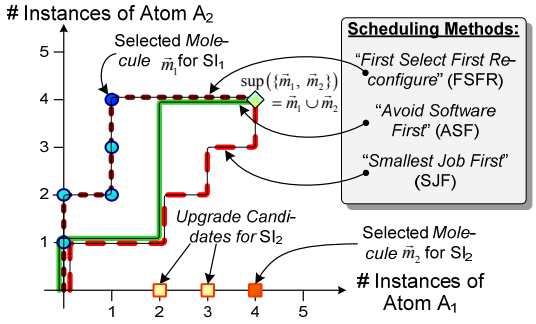


Figure 5: Comparing different Scheduling Methods for two selected Molecules of different SIs

One potential problem of the FSFR schedule is that SI_2 is not accelerated by a *Molecule* until SI_1 is completely upgraded. Therefore, the "*Avoid Software First*" (ASF) scheduler concentrates on first loading an accelerating *Molecule* for all SIs, before following the path of FSFR, as shown in Figure 5. Continuing the idea of loading small *Molecules* first leads to the *"Smallest Job First"* (SJF) schedule. At first, the smallest hardware *Molecule* is loaded for each SI (similar to ASF). Then, for all remaining *Molecule* candidates the number of additional required *Atoms* is determined and the *Molecule* with the minimal additional *Atoms* is selected. If two or more *Molecules* require the same minimal number of additional *Atoms*, then the *Molecule* with the bigger performance improvement is scheduled first.

All three presented schedulers bear certain drawbacks. Either they concentrate on one SI after the other (FSFR & ASF in both phases) or they concentrate on selecting the locally smallest upgrade step. What we need is a scheme that situation-dependent determines, whether it is more beneficial to continue upgrading a certain SI or to switch to a different SI and later on continue the previous SI. Therefore, we need a metric to find the intermediate steps (i.e. *Molecule*) that are the most beneficial ones on a scheduling path up to $\sup(M)$. Figure 6 shows the pseudo code of our proposed "*Highest Efficiency First*" (HEF) scheduling algorithm. The benefit computation for the *Molecule* candidates can be seen in line 20. The performance improvement compared to the currently fastest available/scheduled *Molecule* for the same SI is weighted with the number of expected executions for this SI and relativized with the number of additionally required *Atoms*.

```
1.  // consider all smaller Molecules, see equation (3)
2.  M' ← ∅;
3.  ∀m⃗ ∈ M
4.    ∀o⃗ : o⃗ ≤ m⃗ ∧ o⃗.getSI() = m⃗.getSI()
5.      M' ← M' ∪ {o⃗};
6.  // initialize the bestLatency array
7.  a⃗ ← currentlyAvailableAtoms;
8.  ∀m⃗ ∈ M
9.    m⃗.getSI().bestLatency ← m⃗.getSI().
        getFastestAvailableMolecule(a⃗).getLatency();
10. // schedule the Molecule candidates
11. scheduledList ← ∅;
12. while (M' ≠ ∅) {
13.   // clean Molecule candidates, see equation (4)
14.   ∀m⃗ ∈ M'
15.     if (m⃗ ≤ a⃗ ∨ m⃗.getLatency() ≥ m⃗.getSI().bestLatency)
16.       M' ← M' \ {m⃗};
17.   if (M' = ∅) break;
18.   bestBenefit ← 0;
19.   ∀o⃗ ∈ M'{
20.     benefit ← (o⃗.getSI().getExpectedExecutions() *
          (o⃗.getSI().bestLatency − o⃗.getLatency())) / |a⃗ ▷ o⃗|;
21.     if (benefit > bestBenefit) {
22.       bestBenefit ← benefit; m⃗ ← o⃗;
23.     }
24.   }
25.   // schedule the chosen Molecule
26.   ∀Atoms a_i ∈ (a⃗ ▷ m⃗) scheduledList.push(a_i);
27.   a⃗ ← a⃗ ∪ m⃗;
28.   m⃗.getSI().bestLatency ← m⃗.getLatency();
29. }
30. return scheduledList;
```

Figure 6: Our implemented scheduling method "Highest Efficiency First" (HEF)

5. Evaluation and results

We will now present a detailed analysis of the presented schedulers from Section 4 by benchmarking them with an implementation of the H.264 video encoder and a CIF-video (352x288) with 140 frames (more details about the application are given in [25]). Then, we compare a state-of-the-art reconfigurable computing system [19] with our architectural extension of stepwise *upgrading* the *Molecules* to their final implementation and highlight the speedup that we additionally achieve by our proposed Special Instruction Scheduler. Table 1 summarizes our implemented SIs for the H.264 encoder with the different required types of *Atoms* and the number of available *Molecules*.

	Special Instruction	# *Atom*-types	# *Molecules*
Motion Estimation (ME)	SAD	1	3
	SATD	4	20
Encoding Engine (EE)	(I)DCT	3	12
	(I)HT_2x2	1	2
	(I)HT_4x4	2	7
	MC_4	3	11
	IPred_HDC	2	4
	IPred_VDC	1	3
Loop Filter (LF)	LF_BS4	2	5

Table 1: Implemented SIs of H.264 with number of required Atom-types and number of available Molecules

Figure 7 shows the execution time for encoding 140 frames with different *Atom Container* (AC) quantities. The results for 0 to 4 ACs are omitted for clarity, because their execution time is significantly slower (down to the execution speed of a general-purpose processor in case of zero ACs: 7,403M cycles). A noticeable situation can be seen when seven ACs are available. The performance for the FSFR (*First Select First Reconfigure*), ASF (*Avoid Software First*), and later also SJF (*Smallest Job First*) scheduler degrades when more ACs are added. This is due to the fact, that *bigger Molecules* (i.e. *Molecules* with more *Atoms*) are selected (as more space is available) and this increases the reconfiguration time until the selected *Molecules* are finally configured. Therefore, although these bigger *Molecules* offer the potential for a faster execution, this potential has to be made available by a more sophisticated scheduling scheme. Especially FSFR fails here, as it strictly upgrades one SI after the other. However, from 17 ACs on, FSFR outperforms ASF, as ASF initially spends some time to accelerate *all* SIs, even though some of them are significantly less often executed than others are. The HEF (*Highest Efficiency First*) scheduling does not underlie such drawbacks, as it is able to weight the importance of the *Molecules* independently. The more ACs are available, the clearer the differences between the scheduling methods become apparent.

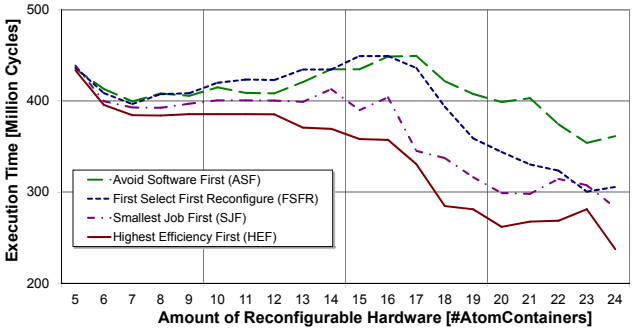


Figure 7: Comparing our proposed scheduling schemes while encoding 140 Frames of a CIF video sequence

In Table 2 we compare the speedup of the worst (ASF) with the best (HEF) scheduler. We have further on analyzed the behavior of state-of-the-art related reconfigurable computing systems, i.e. Molen [19] and OneChip [21]. They both provide a single implementation per SI and thus cannot upgrade during run time. We have simulated this behavior and compared it to our hierarchical SI implementation. Table 2 shows the comparison of the ASF scheduler vs. a Molen-like [19] state-of-the-art reconfigurable system (for a fair comparison, the same hardware accelerators are provided to Molen). The gained performance due to our proposed HEF scheduler further improves the speedup compared to Molen. Although the ASF scheduler already reaches a speedup of up to 1.67x (23 ACs) compared to Molen, our proposed HEF scheduler provides us up to 1.52x (24 ACs) on top, altogether resulting in a speedup of up to 2.38x (24 ACs). In average, our HEF scheduler achieves 1.71x speedup compared to Molen and it is noteworthy that it never performed slower than Molen or any of the other schedulers. It shows that our *RISPP* concept is superior to the state-of-the-art embedded processor paradigm where Special Instructions (SIs) are determined at design time and fix at run time whereas we can gradually upgrade depending on the state of the system. To achieve this we need a more intelligent run-time system, which implies hardware overhead (our HEF implementation requires less area than one AC as discussed below).

#ACs	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
HEF vs ASF	1.00	1.04	1.04	1.06	1.05	1.08	1.06	1.06	1.13	1.18	1.21	1.26	1.36	1.48	1.45	1.52	1.51	1.39	1.26	1.52
ASF vs Molen	1.08	1.07	1.12	1.12	1.21	1.22	1.26	1.38	1.39	1.34	1.40	1.36	1.41	1.50	1.54	1.56	1.54	1.58	1.67	1.57
HEF vs Molen	1.09	1.12	1.16	1.19	1.28	1.31	1.34	1.46	1.57	1.58	1.70	1.70	1.92	2.22	2.23	2.38	2.32	2.21	2.11	2.38

Table 2: Speedup due to our HEF scheduling and speedup compared to a Molen-like [19] state-of-the-art reconfigurable computing system (ACs: Atom Containers)

Let us analyze the detailed scheduling behavior of HEF, shown in Figure 8 for 10 ACs. It shows the first two hot spots ("*Motion Estimation"* and "*Encoding Engine"* as illustrated in Figure 1) executed for one frame. The lines show the latencies for four SIs (logarithmic scale) and thus the immediate scheduler decision. Whenever a latency line decreases, the *Atoms* to upgrade the *Molecule* just finished loading. The bars show the resulting SI execution frequency for periods of 100K cycles, thus showing the performance improvement due to the scheduling.

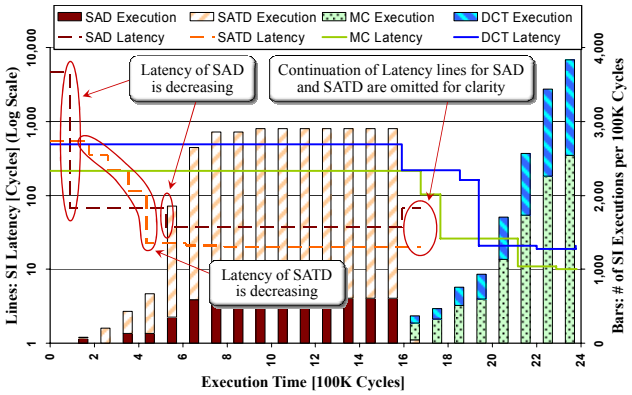


Figure 8: Detailed analysis of HEF scheduler for the first two hot spots (ME and EE) of one encoded frame

We have implemented and run a hardware platform based on an HW-AFX-FF1152-200 board (see Figure 9) for a Xilinx xc2v3000-6 FPGA. As our *Atoms* can be implemented in rather small modules of the FPGA (avg. 421 Slices, see Table 3), the partial Bitstream requires in average only 60,488 Bytes (due to FPGA-specific constraints we had to use four CLB rows). This results in an average reconfiguration time of 874.03 µs [23] (for 66 MB/s reconfiguration bandwidth via the SelectMap/ICAP [6] interface). Finally, the HEF scheduler (the focus of this paper) is implemented in a finite state machine with 12 states. The part with the largest computational effort within the scheduler is to calculate the *benefit* of a *Molecule* candidate (line 20 in Figure 6). First, we have pipelined the *benefit* computation. Secondly, we avoided the expensive (concerning performance, area etc.) division by exploiting the fact that we only need to be able to *compare* two *benefit* values but we do not need the actual values. Accordingly, we were able to change the equation for the *benefit* computation from $(a \cdot b)/c > (d \cdot e)/f$ to $(a \cdot b) \cdot f > (d \cdot e) \cdot c$, as we know that the values for c and f (i.e. the number of *additionally* required *Atoms*) are always bigger than zero. The synthesis results are given in Table 3. HEF requires only 3.83% of the available slices (only 1.30x more slices than the average *Atom* size) and would therefore fit into one AC (1024 slices). Its clock delay does not affect the critical path of our current prototype.

Characteristics	Our HEF scheduler	Avg. *Atom*
# Slices	549	421
# LUTs	915	839
# FFs	297	45
#MULT18X18	5	0
Gate Equivalents	30,769	6,944
Clock delay [ns]	12.596	1.284

Table 3: Hardware implementation results of our HEF scheduler

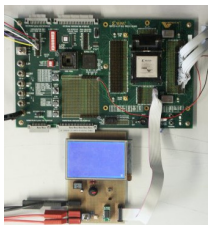


Figure 9: Hardware evaluation platform

6. Conclusion

As shown, the whole system including the scheduler part has been evaluated in simulation and measured in actual hardware implementation. An H.264 video encoder has been analyzed in detail to explore the new concept (though it is certainly not limited to it). Deciding upon the composition of a Special Instruction (SI) during run time is even farther beneficial compared to the state-of-the-art reconfigurable embedded processor approach (e.g. Molen [19]) when it comes to hard-to-predict application behavior. We measured a 2,38x increase over a Molen-like reconfigurable system with fixed SI composition and schedule. The inherent lower performance of the reconfigurable hardware is compensated by the high degree of *Atom*-level parallelism and the dynamically changeable *Molecule*-level parallelism.

7. References

[1] ARCtangent processor. ARC International. (www.arc.com)

[2] ASIP Meister. (http://asip-solutions.com)

[3] CoWare Inc, LISATek. (www.coware.com)

[4] Target Compiler (http://www.retarget.com)

[5] Stretch processor (www.stretchinc.com)

[6] Xilinx Corp. "XAPP 290: Two Flows for Partial Reconfiguration: Module Based or difference Based" www.xilinx.com

[7] Xtensa processor, Tensilica Inc.: www.tensilica.com

[8] ITU-T and ISO/IEC JVT "Advanced video coding for generic audiovisual services", ITU-T Rec. H.264 and ISO/IEC 14496-10:2005 (E) (MPEG-4 AVC), March 2005

[9] J. Henkel "Closing the SoC Design Gap", IEEE Computer Volume 36, Issue 9, September 2003, pp. 119-121

[10] A. Chattopadhyay et al. "Design Space Exploration of Partially Re-configurable Embedded Processors", DATE'07, pp. 319-324

[11] H. P. Huynh, E. Sim, T. Mitra "An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization", CASES 2007, pp. 135-144

[12] N. Clark H. Zhong S. Mahlke "Processor Acceleration Through Automated Instruction Set Customization", MICRO-36 2003, pp. 129-140

[13] N. Cheung, J. Henkel, S. Parameswaran "Rapid Configuration & Instruction Selection for an ASIP: A Case Study", DATE 2003, pp. 802-807

[14] K. Atasu, L. Pozzi, P. Ienne "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints", DAC 2003, pp. 256-261

[15] F. Sun, A. Raghunathan, S. Ravi, N. K. Jha "A scalable application specific processor synthesis methodology", ICCAD 2003, pp. 283-290

[16] K. Compton, S. Hauck "Reconfigurable computing: a survey of systems and software", ACM Computing Surveys 2002, pp. 171-210

[17] M. Ullmann et al. "On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities" FPL 2004, pp. 454–463

[18] F. Bouwens, M. Berekovic, A. Kanstein, G. Gaydadjiev "Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array", ARC 2007, pp. 1-13

[19] S. Vassiliadis et al. "The MOLEN polymorphic processor", IEEE Transaction on Computers, Issue 11, 2004, pp. 1363-1375

[20] F. Barat, R. Lauwereins "Reconfigurable Instruction Set Processors: A Survey", in Rapid System Prototyping 2000, pp. 168-173

[21] J. E. Carrillo, P. Chow "The Effect of Reconfigurable Units in Superscalar Processors", FPGA 2001, pp. 141-150

[22] R. Lysecky, G. Stitt, F. Vahid "Warp Processors", ACM Trans. on Design Autom. of Electronic Systems Vol. 11, 2006, pp. 659-681

[23] L. Bauer, M. Shafique, S. Kramer, J. Henkel "RISPP: Rotating Instruction Set Processing Platform", DAC'07, pp.791-796

[24] L. Bauer, M. Shafique, D. Teufel, J. Henkel "A Self-Adaptive Extensible Embedded Processor", SASO'07, pp. 344-377

[25] M. Shafique, L. Bauer, J. Henkel "An Optimized Application Architecture of the H.264 Video Encoder for Application Specific Platforms", ESTIMedia 2007, pp. 119-142