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 adaptability 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, making 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 less 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 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.

Reconfigurable computing may address this problem of inefficiently utilized resources through reconfigurable hardware ([15], [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 hardware1. 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
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.

A bridging 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 system (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 application3 (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 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 Clip) 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 simplest implementation of an SI is without any accelerating Atoms, only using

---

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.

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 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:

1) Controlling the execution of SIs
2) Observing and adapting to changing/varying constraints
3) 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 \( \tilde{m}_i \) was selected to implement a required SI. This SI may also be implemented (with reduced performance) by the Molecules \( \tilde{m}_i \) or \( \tilde{m}_j \). A good scheduler should exploit the potential for upgrading from one Molecule to a faster one until the selected Molecule is finally composited. 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

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 \((\mathbb{M}, \mathcal{S}, \omega)\), where \(\mathbb{M}\) is the set of all Molecules and \(n\) is the number of different available Atom-types. For convenience we consider \(m_i \in \mathbb{M}\) as Molecules with \(m_0 = (m_1, ..., m_n)\) where \(m_i\) describes the desired number of instances of Atom \(i\) to implement the Molecule. The operator \(\cup: \mathbb{M} \times \mathbb{N} \to \mathbb{M}\) with \(m_i \cup \omega = \tilde{p}\); \(p_i = \max\{m_i, o_i\}\) describes a Meta-Molecule which contains the Atoms required to implement both \(m_i\) and \(\omega\). We name the resulting Molecule \(\tilde{p}\) 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 \((\mathbb{N}, \omega)\) is an Abelian semi-group. The same is true for \((\mathbb{M}, \omega)\) with \(\cap: \mathbb{M} \times \mathbb{N} \to \mathbb{M}\) being defined as \(m_i \cap \omega = \tilde{p}\); \(p_i = \min\{m_i, o_i\}\) with the neutral element (\(\maxInt, ..., \maxInt\)). The relation \(\preceq\) is defined to be true iff \(\forall i \in [1..n]: m_i \leq o_i\). As \(\preceq\) is reflexive, anti-symmetric, and transitive, \((\mathbb{N}, \preceq)\) is a partially ordered set. For a set of Molecules \(M \subseteq \mathbb{M}\), the supremum is defined as \(\sup M := \bigcup_{m \in M} m\). 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. \(\forall m_i \in M: m_i \preceq \sup M\). The infimum is correspondingly defined as \(\inf M := \bigcap_{m \in M} m\). The infimum from \(M\) is a Meta-Molecule with the meaning of containing those Atoms that are collectively needed for all Molecules of \(M\). As any subset \(\emptyset \neq M \subseteq \mathbb{M}\) has a well-defined supremum and infimum, \((\mathbb{N}, \preceq)\) 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 \(|m| := \sum_{i=1..n} m_i\), i.e. the total number of Atoms that are required to implement \(m_i\). To consider already configured Atoms we define the function

\[ \triangleright: \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+; \quad m \triangleright \omega \equiv \tilde{p}; \quad p_i \equiv \begin{cases} 0 \cdot m_i & \text{if } o_i - m_i \geq 0 \\ o_i - m_i & \text{else} \end{cases} \]

The created Meta-Molecule \(\tilde{p}\) contains the minimum set of Atoms that additionally have to be offered to implement \(\omega\), assuming that the Atoms in \(m_i\) are already available.

4.2. Defining the Atom Scheduling problem

The input to the scheduling is a set \(M = \{|m_i|\}\) of all Molecules \(m_i \in \mathbb{N}\) 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 \(i\)-Atom of \(\sup M\) as \(x_i\), i.e. \(\sup M = [x_1, x_2, ..., x_n]\). We define \(N_A := |\sup M| = \sum x_i\) 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. \(N_A \leq \#ACs\). We define so called
Unit-Molecules (UM) to represent our elementary Atoms: \( \tilde{u} = (1,0,...,0) \), \( \tilde{u} = (0,1,0,...,0) \), ..., \( \tilde{u} = (0,...,0,1) \). With these definitions, we can now define a loading sequence for Atoms as a scheduling function:

\[
\text{SF: } [1,k] \rightarrow \text{UM} := \{ \tilde{u}_1, \ldots, \tilde{u}_n \}
\]

(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 \(\text{SF}(j)\) is started. The scheduling function for the better schedule in Figure 4 is \(\text{SF}(1) = \tilde{u}_1, \text{SF}(2) = \tilde{u}_2, \text{SF}(3) = \tilde{u}_3, \text{SF}(4) = \tilde{u}_4, \text{SF}(5) = \tilde{u}_5, \text{SF}(6) = \tilde{u}_6\). 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).

\[
\forall i \in [1,n]: \left\lfloor \frac{m}{|\text{SF}(j)| = \tilde{u}_i} \right\rfloor = x_i
\]

(2)

The condition (2) ascertainment, that SF considers each \(\tilde{u}_i\) in the correct multiplicity \(x_i\), which guarantees, that \(\bigcup_{j \in \text{SF}(i)} \tilde{u}_i = m\). 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 \(\tilde{a}\) denotes the already available/scheduled Atoms and \(m\) the Molecule that shall be scheduled next, then the additionally required Atoms \(\tilde{a} \cup m\) 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 \(\tilde{a} \cup m\) Molecules \(M'\) that may implement the same SIs as the selected Molecules \(M\).

\[
M' = \bigcup_{\tilde{a} \cup m} \tilde{a} \cup m \leq m \cdot \text{getSI}(\tilde{a} \cup m) = m \cdot \text{getSI}(\tilde{a})
\]

(3)

These Molecule candidates \(M'\) are all possible intermediate steps that might be considered on a schedule up to \(\text{sup}(M)\). 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 \(\tilde{m}_3 = (1,3)\) with a worse latency than \(\tilde{m}_2 = (2,2)\) (the latency denotes the number of cycles that are required for a single execution of the corresponding Molecule). After \(\tilde{m}_3\) is composed by the schedule with the dashed line (Figure 4), \(\tilde{m}_3\) is still unavailable and it does not offer a latency improvement. Nevertheless, \(\tilde{m}_3\) may be beneficial depending on the initially available Atoms \(\tilde{a}\) if \(\langle \tilde{a} \uparrow \tilde{m}_3 \rangle \leq \langle \tilde{a} \uparrow \tilde{m}_2 \rangle\) (e.g. for \(\tilde{a} = (0,3)\)). Therefore, we cannot assume that Molecules like \(\tilde{m}_3\) are removed at compile time. We instead clean the list of Molecule candidates from \(M^*\) to \(M^*\) for the currently available or scheduled Atoms \(\tilde{a}\) before we schedule the next Molecule.

\[
M^* = \{ m \in M^*: \langle \tilde{a} \uparrow m \rangle > 0 \wedge m \cdot \text{getLatency} < \text{getFastestAvailable-Molecule mentionedLatency} \}
\]

(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. (A1, A2). The two dark-filled Molecules (circle for SI1, and square for SI2) are selected Molecules for addressing the current hot spot. The lighter filled Molecules are intermediate upgrade possibilities for the two SIs.
We will now present a detailed analysis of the presented schedulers. It shows that our HEF (Highest Efficiency First) scheduler further improves the speedup compared to Molen. Although the ASF scheduler already reaches a speedup of up to 2.38x (24 ACs). In average, our HEF scheduler further improves the speedup compared to Molen. Table 2 shows the comparison of the ASF scheduler vs. a Molen-like [19] state-of-the-art reconfigurable computing system (for a fair comparison, the same hardware accelerators are used). 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 computing 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 fixed at run time whereas we can gradually upgrade depending on the state of the system.

In Table 2 we compare the speed up 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 computing 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 fixed 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).

<table>
<thead>
<tr>
<th>#ACs</th>
<th>HEF vs ASF</th>
<th>ASF vs Molen</th>
<th>HEF vs Molen</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>+1.04</td>
<td>+1.04</td>
<td>+1.04</td>
</tr>
<tr>
<td>6</td>
<td>+1.06</td>
<td>+1.05</td>
<td>+1.05</td>
</tr>
<tr>
<td>7</td>
<td>+1.08</td>
<td>+1.06</td>
<td>+1.06</td>
</tr>
<tr>
<td>8</td>
<td>+1.10</td>
<td>+1.08</td>
<td>+1.08</td>
</tr>
<tr>
<td>9</td>
<td>+1.12</td>
<td>+1.09</td>
<td>+1.09</td>
</tr>
<tr>
<td>10</td>
<td>+1.14</td>
<td>+1.12</td>
<td>+1.12</td>
</tr>
<tr>
<td>11</td>
<td>+1.16</td>
<td>+1.14</td>
<td>+1.14</td>
</tr>
<tr>
<td>12</td>
<td>+1.18</td>
<td>+1.16</td>
<td>+1.16</td>
</tr>
<tr>
<td>13</td>
<td>+1.20</td>
<td>+1.18</td>
<td>+1.18</td>
</tr>
<tr>
<td>14</td>
<td>+1.22</td>
<td>+1.20</td>
<td>+1.20</td>
</tr>
<tr>
<td>15</td>
<td>+1.24</td>
<td>+1.22</td>
<td>+1.22</td>
</tr>
<tr>
<td>16</td>
<td>+1.26</td>
<td>+1.24</td>
<td>+1.24</td>
</tr>
<tr>
<td>17</td>
<td>+1.28</td>
<td>+1.26</td>
<td>+1.26</td>
</tr>
<tr>
<td>18</td>
<td>+1.30</td>
<td>+1.28</td>
<td>+1.28</td>
</tr>
<tr>
<td>19</td>
<td>+1.32</td>
<td>+1.30</td>
<td>+1.30</td>
</tr>
<tr>
<td>20</td>
<td>+1.34</td>
<td>+1.32</td>
<td>+1.32</td>
</tr>
<tr>
<td>21</td>
<td>+1.36</td>
<td>+1.34</td>
<td>+1.34</td>
</tr>
<tr>
<td>22</td>
<td>+1.38</td>
<td>+1.36</td>
<td>+1.36</td>
</tr>
<tr>
<td>23</td>
<td>+1.40</td>
<td>+1.38</td>
<td>+1.38</td>
</tr>
<tr>
<td>24</td>
<td>+1.42</td>
<td>+1.40</td>
<td>+1.40</td>
</tr>
</tbody>
</table>

Figure 7 shows the execution time for encoding 140 frames with 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 computing 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 fixed 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).

![Figure 7: Comparing our proposed scheduling schemes while encoding 140 Frames of a CIF video sequence](image-url)
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 Molecular 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](image1)

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 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.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Our HEF scheduler</th>
<th>Avg. Atom</th>
</tr>
</thead>
<tbody>
<tr>
<td># Slices</td>
<td>549</td>
<td>421</td>
</tr>
<tr>
<td># LUTs</td>
<td>915</td>
<td>839</td>
</tr>
<tr>
<td># FFs</td>
<td>297</td>
<td>45</td>
</tr>
<tr>
<td>#MULT1T18X18</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Gate Equivalents</td>
<td>30,769</td>
<td>6,944</td>
</tr>
<tr>
<td>Clock delay [ns]</td>
<td>12.596</td>
<td>1.284</td>
</tr>
</tbody>
</table>

Table 3: Hardware implementation results of our HEF scheduler

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