Mediaprocessor Programming Interface to Increase the Portability of Mediaprocessor Software

Michael S. Grow, Siddhartha Sikdar, Donglok Kim, and Yongmin Kim
Image Computing Systems Laboratory
Departments of Electrical Engineering and Bioengineering
University of Washington, Seattle, WA 98195-2500

ABSTRACT

The architecture of mediaprocessors has become increasingly sophisticated to accommodate the need for more performance in processing various media data. However, due to the inability of mediaprocessor compilers to fully detect the parallelism available in a program and maximize the utilization of the mediaprocessor’s on-chip resources, C intrinsics, which are hints to the compiler on which assembly instructions to use, have been employed to achieve better performance. Nonetheless, these intrinsics are mediaprocessor-dependent, thus limiting the portability of mediaprocessor software. To help increase the portability of mediaprocessor software, we have developed a Mediaprocessor Programming Interface (MPI), which translates one set of C intrinsics into another. In many cases, the translated code for the target mediaprocessor has similar performance to the code developed with native intrinsics. We believe that the MPI can facilitate the reuse of mediaprocessor software as well as the development of mediaprocessor-independent software.

Keywords: Mediaprocessor, portability, MAP1000, MAP-CA, intrinsics, TriMedia

1. INTRODUCTION

Increased processing and computing needs in commercial and consumer products, such as digital color copiers, cellular phones, digital TVs, digital cameras/camcorders and set-top boxes, have led to recent advances in algorithms and processors aimed for multimedia specific computations. One such advance is the development of mediaprocessors [1], such as Philips TriMedia, Hitachi/Equator Technologies MAP and Texas Instruments TMS320C64X. Mediaprocessors combine the computational power of high-end DSPs with various I/O capabilities and auxiliary units in a programmable architecture to offer a flexible and adaptable solution for multimedia applications [2]. Due to the increasing complexity and lack of flexibility of ASICs and other hardwired components, these mediaprocessors are becoming more attractive than ASICs to fulfill the processing and computing needs of multimedia applications.

In consumer products, however, mediaprocessors have not had the widespread success that they have had in professional and commercial products, e.g., medical imaging and machine vision [2]. Since many consumer companies have long depended on hardwired solutions in developing their products, changing to a mediaprocessor-based approach would cause a company to have to change their product development paradigm completely. For instance, unfamiliarity in programming mediaprocessors has been a factor limiting the success of mediaprocessors in consumer products. In many cases, compilers currently have difficulty in fully exploiting the mediaprocessor resources and parallelism available in a program to achieve optimal performance, which forces developers to use low-level assembly programming, thus lowering the programming productivity. Furthermore, while mediaprocessors provide the flexibility and fast time-to-market that companies desire, developing optimized applications on mediaprocessors currently is time-consuming and requires extensive retraining of engineers.

Many mediaprocessors have been designed in order to exploit unique features in multimedia and image processing for higher performance, for example, predictable data access patterns and data-level parallelism, i.e., the computation can be performed on multiples of small data units. Partitioned operations are widely available in mediaprocessors for increased parallelism. Also, to reduce memory access latency, efficient data movement engines are incorporated.

However, current mediaprocessor compilers typically have difficulty in (1) programming the data flow of an algorithm to minimize the total I/O time and (2) taking advantage of mediaprocessor-specific assembly instructions that operate on
partitioned data. Many mediaprocessors contain intelligent direct memory access (DMA) engines for handling the data flow of an algorithm. Nonetheless, these DMA engines (and/or their programming interfaces) vary from one mediaprocessor to the next, thus making the data flow component of software mediaprocessor-specific. Choosing right assembly instructions is very challenging for a compiler since it is difficult to determine when to use partitioned operations from generic C code. Various saturation and rounding modes make the task even more difficult. C intrinsics, which look like normal C function calls, tell the compiler exactly what assembly instructions to use. Since C intrinsics mimic assembly instructions, their use makes the software more mediaprocessor-specific. This concern, developing software that runs on only one mediaprocessor, has been a critical reason for companies to hesitate in adopting mediaprocessors since the software investment could be lost in trying to migrate to a new and more powerful mediaprocessor.

To help companies protect their software investment and facilitate the rapid development of software for newly-introduced mediaprocessors, we have developed a mediaprocessor programming interface (MPI), which translates the intrinsics from one mediaprocessor into those for another mediaprocessor. Using this tool, the core loops of application code that typically use C intrinsics for enhanced performance can be seamlessly translated from one mediaprocessor platform to another. The performance of code generated by the MPI translator is expected to be somewhat lower than that of code originally developed using the native intrinsics of a mediaprocessor. Nevertheless, the MPI could significantly reduce porting efforts as well as lead to faster deployment of software for mediaprocessor-based products because the programmer can use translated code from an earlier mediaprocessor platform as a starting point. To demonstrate the feasibility and usefulness of the MPI, we have translated image computing functions previously developed for several mediaprocessors into functions to run on other mediaprocessors. In this paper, we will describe how the C intrinsics are translated and compare the performance of MPI-generated code with that of code developed specifically for the same mediaprocessor.

2. TRANSLATION

2.1 Goals of translation

The first goal of intrinsic translation is to generate code that has comparable performance to code that is developed specifically for the target mediaprocessor. The second goal is to generate easy-to-read code so that it can be easily improved by hand when necessary. Specifically, the generated code should be properly indented as well as contain the comments from the original code to make it easy to understand and maintain [3,4]. The last goal is to confine all of the mediaprocessor-specific information, e.g., data type names, intrinsic names and special compiler directives, to a translation table, thus keeping the translator mediaprocessor-independent. As a result, to translate for a different mediaprocessor, only the translation table has to be modified.

2.2 The translator

Figure 1 shows the structure of the MPI translator. The first stage of the translator is a preprocessor, which includes header files and expands macros in the input code. The preprocessing stage is necessary so that the translator can see the definitions of the user-defined types used in the input code. Furthermore, any macros that are used in the input code should be expanded in case they contain intrinsics that need to be translated. In the next stage, the translation table and the input code are parsed to obtain the information that the translator needs. The parsing of the input code is similar to the parsing routine in a common C compiler [5]. Specifically, the source program is converted into a string of tokens, which is checked for syntactic correctness and then converted into a syntax tree. Also, a symbol table is created to keep track of the data type of each variable. After the syntax tree and symbol table are created, the actual translation occurs. Once translation is complete, the code generator prints the output code into a file.

2.3 Translation table

2.3.1 Structure of the translation table

The translation table contains the information that the translator needs to convert one set of intrinsics into another. The first section of the translation table contains the names of the mediaprocessor-dependent header files that need to be included/removed from the target code. The next section provides a data type mapping. For example, in the translation table for converting Pentium III MMX C intrinsics into MAP C intrinsics, this part would specify that the Pentium III MMX data
type of __m64 should be converted to the 64-bit MAP data type n64. After the data type mapping, the translation table contains a list of options that control the translator's operation. For example, in the table for translating code containing Pentium III MMX C intrinsics into MAP C intrinsics, these options tell the translator that the target processor is a 64-bit media processor, which supports performance enhancing compiler directives for loop preconditioning, loop unrolling and pointer disambiguation through use of the "restrict" keyword, but lacks an instruction to request cache prefetching. The last part of the translation table consists of a mapping between every intrinsic offered by the source processor and the most optimal and accurate representation of that intrinsic in the target processor, either as a single intrinsic or as a combination of several intrinsics.

2.3.2 Intrinsic mappings

Developing the mappings between the intrinsics is an important part in setting up the translation process since an inefficient mapping could likely result in generated code with poor performance. The mapping of an intrinsic on a target mediaprocessor depends on the argument types and saturation/rounding mode options of the intrinsic. Therefore, multiple mappings are typically needed for each intrinsic (one for each possible combination of argument types and user options).

Most mediaprocessors have instructions with built-in saturation. In some cases, however, saturation must be done explicitly, which is quite inefficient since it requires expanding the data partitions to a larger width for the computation and saturation and then compressing the result back to the original width. In saturated addition of 8-bit images, for example, if the target processor does not have an instruction that automatically saturates after adding two 8-bit partitioned data, each 8-bit partition must be expanded to 16 bits for the addition to prevent the values from overflowing. Following the addition, partitions with a value greater than 255 are clipped to 255. Lastly, the values need to be compressed back down to 8 bits.

Fixed-point rounding modes also add to the complexity of developing the intrinsic mappings. For mediaprocessors that do not support these rounding modes, it is necessary to develop a sequence of instructions that emulate these rounding modes. Figure 2 shows how the MAP C intrinsics for right shifting 64-bit values with 8-bit partitions are emulated using other MAP C intrinsics without fixed-point rounding modes.

2.3.3 Translating across architectures of different widths

Most mediaprocessors contain 64-bit architectures, and future architectures are likely to be wider than 64 bits. However, a few, such as the TriMedia, have 32-bit architectures. Fortunately, when translating mediaprocessor software to an architecture with a different width, no additional changes need to be made to the translator. The translation table can take care of the change in architecture width with relative ease as long as the set of intrinsics for the target mediaprocessor contains intrinsics for merging into and extracting partitions out of wider values. For example, the MAP includes a couple of C intrinsics (hmpv_gethi_32, and hmpv_getlo_32) for accessing the high/low 32 bits of a 64-bit value. If these kinds of C intrinsics are not available, they can be emulated using generic C.

In translating from a wide architecture to a narrower one, intrinsics operating on wide data can be replaced with a series of intrinsics that operate on the data a portion at a time. For example, when translating from a 64-bit to a 32-bit architecture, an intrinsic that adds two 64-bit values with 8-bit partitions can be replaced with two intrinsics that each add 32-bit values with 8-bit partitions. Translating from a narrow architecture to a wider one, however, is much more difficult to do in an efficient manner. A simple method is to append zeros to the data before processing. Nonetheless, this method is inefficient since it only uses a portion of the available architecture. A more efficient method will be introduced in Section 3.3.

To simplify the translation from a wide architecture to a narrower one further, structure types could be created for representing wide data on narrow mediaprocessors, for example, 64-bit data on a 32-bit mediaprocessor. The fields of the MPI types can be used directly in the translation table, thus eliminating the extra intrinsics to extract the individual partitions. For example, if a type "u64_32" is created as a union of a 64-bit type (field named "whole") and a structure (field named "half") containing the fields "hi" and "lo" (for the high and low 32 bits of the 64-bit value, respectively), an intrinsic in the translation table could either use the "half.lo" and "half.hi" fields to access only 32 bits or the "whole" field to access all 64 bits at once. This type of structure works well on architectures with multiple register sizes. For example, on the MAP the 64 32-bit registers can be treated as 32 64-bit registers. Nonetheless, this approach may also lead to inefficient performance. One example is on the Pentium III where the 64-bit MMX registers are independent of the 32-bit general-purpose registers.
Therefore if a 32-bit portion of a 64-bit value is accessed, it must store the 64-bit value to memory and load the 32-bit portion into the 32-bit general-purpose register bank. Table 1 contains a list of MPI types used in the translator.

2.4 Details of translation

Finding intrinsics in the input code involves looking at each function call to see if the name of the called function is an intrinsic or a saturation/rounding mode option specifier. If the name of the called function is an option specifier, it is recorded to help with the intrinsic selection process. The argument of the option specifier is then checked to see if it is another option specifier or an intrinsic. The selection of an intrinsic or a group of intrinsics to insert in place of the old intrinsic is based on the following parameters: intrinsic name, number of arguments, data type of arguments, and saturation and rounding mode options. When using the data type of the arguments for selecting the target intrinsic(s), structural equivalence [5] is used. In other words, the translator will consider two data types with different names to be the same if they have the same structure.

In addition to translating intrinsics, the translator is responsible for changing data types as well. In some cases, the printing stage of the translator can handle this task by looking up the data type in the data type mapping section of the translation table and getting the corresponding type for the target processor. In other cases, when multiple types offered by the target processor can be mapped to a single type offered by the source processor, this task is a bit more difficult. In this case, the data types are thus translated at the same time as the intrinsics so that an intrinsic that uses a variable can help distinguish the appropriate type for that variable.

3. RESULTS AND DISCUSSION

3.1 Performance results

Tables 2 and 3 show the performance results of the MPI for a set of functions. Each of the functions operates on 512 x 512 grayscale images except for binary dilate, which operates on a 512 x 512 binary image. To obtain the results for the “Generic C” columns, we ran a generic C version of the function, which does not contain any intrinsics, on the respective processor. The “Native C Intrinsics” columns represent functions that we have developed for the respective mediaprocessor using the native intrinsics of the mediaprocessor. The “TriMedia” columns represent functions that we developed for the TriMedia and then translated for the MAP. The functions in the “Native C Intrinsics” column of Table 2 included code to use the Data Streamer to minimize the influence of data flow from the results. There is a large performance difference between the results in the “Native C Intrinsics” and “TriMedia” columns, the reason of which was described in Section 2.3.3.

Table 3 shows the performance of functions running on the TriMedia TM1100 cycle-accurate simulator at 133 MHz. The results in the column labeled “TriMedia” represent functions that we developed for the TriMedia and then translated for the MAP. These results were pretty close to the results in the “Native C Intrinsics” column even though the original functions were developed for a mediaprocessor with a different width architecture.

3.2 Universal C intrinsics

In addition to translating C intrinsics for a specific mediaprocessor, the MPI translator can also translate to/from mediaprocessor-independent intrinsics (universal C intrinsics). In selecting a test set of universal C intrinsics, we studied several image computing functions from the MAP UWICL for the MAP mediaprocessor, which was developed by the University of Washington [6]. In addition, we carefully evaluated the instruction sets from a collection of mediaprocessors (e.g., MAP, TriMedia, Pentium III, and TMS320C64X). The criteria we used in selecting intrinsics are: (1) frequency of usage in image computing functions from the MAP UWICL, (2) importance in key image computing functions and (3) frequency of inclusion in mediaprocessor instruction sets. Figure 3 contains some code fragments to illustrate the difference between universal C intrinsics and MAP intrinsics. Initially, we were using the universal C intrinsics as an intermediate representation for translating one set of intrinsics into another, but we later eliminated an intermediate representation. The MPI types in Table 1 were used as an intermediate representation for translating between mediaprocessor-specific data types.
as well. The universal C intrinsics could be used for developing mediaprocessor-independent software. A section of the
translation table for translating universal C intrinsics into MAP C intrinsics is shown in Figure 4.

3.3 Optimization

The translator does not always generate code with performance similar to code developed with native intrinsics. Code
generation from a narrower architecture to a wider architecture is not always optimal because it is difficult to fully utilize the
wider data path. For example, the translator produces non-optimal code when translating 32-bit TriMedia intrinsics into the
64-bit MAP intrinsics, as seen in Table 2. Another case is when an architecture does not have an intrinsic with the correct
partition size for an operation. A common example is when 8-bit data have to be expanded to 16 bits for processing and then
compressed back to 8 bits. If consecutive intrinsics present this situation, a redundancy would result from compressing and
then expanding the data back and so on. It would be more efficient to expand the data once, do all the processing, and then
compress it back down to the desired precision. Nonetheless, the current translator is not sophisticated. An additional
optimization stage could be used.

An optimization stage would require more knowledge of what the input code is doing than the translator does. For example,
the optimization stage would need to know which instructions are not being used efficiently. The "knowledge" of the
optimization stage can be improved by defining a new set of data types (optimizer types) that contain the history of the data
they contain. For example, when 8-bit data values are expanded to 16 bits, "double-width byte" would be a better
representation than "half-word" or "short". When a pointer to 32-bit values is cast as a pointer to 64-bit values, the resulting
type should be "double-width word". These types would allow the optimization stage to know when an intrinsic is being used
inefficiently. For example, the Pentium III does not have an intrinsic for multiplying 64-bit data with 8-bit partitions.
Therefore, the data are expanded to 16 bits for the multiplication. When the function is translated to MAP intrinsics, the use
of optimizer types would give the optimizer the ability to see that the code is using an intrinsic for multiplying 16-bit
partitions to multiply 8-bit partitions. The optimization stage would then switch the intrinsic for multiplying 16-bit partitions
to an intrinsic for multiplying 8-bit partitions (assuming one is available) and remove the intrinsics that expand the arguments
(from 8 to 16 bits) before the multiply intrinsic and compress the result (from 16 to 8 bits) after the intrinsic.

In addition to adjusting partition size, an optimization stage could also make the translation from a narrow to a wide
architecture more efficient. One approach would be to "scale" the code. For example, in translating from a 32-bit to a 64-bit
architecture 32-bit loads/stores would be replaced by 64-bit loads/stores and intrinsics operating on 32-bit partitioned values
would be replaced by their 64-bit counterparts on the target processor. However, determining when and how to scale properly
could be difficult. Another approach could be to unroll a loop containing intrinsics and merge groups of intrinsics into a
single intrinsic. This approach could prove to be difficult as well. Other approaches and/or combinations of approaches will
be researched further.

3.4 Automatic vs. semiautomatic translation

One possible method to improve the results in Tables 2 and 3 is to give the user more control over the translation process,
i.e., semiautomatic translation. For example, the translation table could be expanded to give the user a choice of different
mappings for some intrinsics. One mapping could provide higher accuracy at lower performance while another mapping
could provide higher performance with lower accuracy. Giving the user more control would likely require the development of
a GUI that would let the user step through the translation one intrinsic at a time and select the desired mapping (when
multiple mappings are available). Furthermore, the GUI could let the user assist the translator when conflicts or ambiguities
occur. For example, if there is a variable that is sometimes used for signed 16-bit values and other times for unsigned 16-bit
values, the user could help the translator determine on a case-by-case basis whether the values should be treated as signed or
unsigned.

Semiautomatic translation would give the user more control and could potentially give better performance results, but it
would make translation more complicated and time-consuming. Furthermore, it could require more knowledge about the
target mediaprocessor architecture than automatic translation would. Therefore, the GUI could let the user choose between
automatic and semiautomatic translation. Intrinsics in the translation table would then have one default mapping for
automatic translation. The alternative mappings would be offered if the user decides to use semi-automatic translation. The
mapping that provides the highest accuracy should be the default mapping since correctness is often more important than
performance.
3.5 Universal Data Flow Code Generator

In recent years, the computation core of mediaprocessors has greatly improved in terms of the number of operations that can be performed per instruction cycle. Traditionally compute-intensive tasks can be performed very efficiently with the help of the improved instruction set architecture of modern mediaprocessors. However, the memory bandwidth has not been able to keep up with this processing throughput. As a result, the overall performance of multimedia applications is limited by the external memory access latency in many cases. Therefore, any tool that translates mediaprocessor software from one platform to another should be able to handle the data flow in addition to the core computation based on intrinsics.

Data flow programming in mediaprocessors is typically handled in two ways: programmable DMA (direct memory access) engines and cache prefetching [7]. Implementing data flow using a DMA engine involves software development around the core computation tight loop. Nonetheless, this software is mediaprocessor-specific. On the other hand, due to similarities between DMA engine architectures it could be possible to create an abstraction layer that could be used to generate DMA code for different mediaprocessors. Specifically, some of the similarities include the use of descriptors, including data address and transfer size information for performing the DMA and APIs for starting/stopping the DMA transfer and polling for transfer completion. The abstraction layer could consist of generic macros for DMA descriptor generation, e.g., DMA_start, DMA_stop and DMA_poll. Using this abstraction layer, it is possible to generate data flow code for different mediaprocessors with the help of a GUI-based tool that would give the user an interface to compose the data flow around the computation tight loops. A data flow code generator (DFCG) designed specifically for the MAP has already been developed and could provide a starting point for developing a Universal Data Flow Code Generator (UDFCG) [8]. Figure 5 shows how a UDFCG would be combined with the MPI.

Software using cache prefetch instructions is difficult to translate for another mediaprocessor that uses a DMA engine, and vice versa. It might be possible to translate software between two mediaprocessors that only use cache prefetching. The cache prefetch instructions could be treated as intrinsics, which can then be translated through the MPI. However, if the cache architectures of the two mediaprocessors are widely different or if the prefetching mechanisms are different, this approach may not work.

4. CONCLUSION

In this paper, we introduced a Mediaprocessor Programming Interface (MPI), which facilitates the translation of mediaprocessor software by translating the native C intrinsics of one mediaprocessor into the intrinsics for another mediaprocessor. We have translated previously-developed image computing functions for various mediaprocessors into functions that will run on different mediaprocessors. We have demonstrated that the generated code in the majority of the cases has performance comparable to code developed using native intrinsics. For the cases where the performance is lower than that of code developed using native intrinsics, methods for further improving the performance of generated code were discussed.

The main difficulty of intrinsic translation lies in developing optimal representations of the source intrinsics in the target processor for the translation table. However, once the translation tables have been developed, translating functions from one mediaprocessor into functions that run on other mediaprocessors is a straightforward process. Even if the translator does not always generate the most optimal code for a specific architecture, the MPI could lead to faster deployment and increase the lifetime of mediaprocessor software, thus helping a company protect its software investment.
REFERENCES


Table 1. MPI Types

<table>
<thead>
<tr>
<th>Type</th>
<th>n64</th>
<th>p64u32</th>
<th>p64u16</th>
<th>p64u8</th>
<th>s64</th>
<th>p64s32</th>
<th>p64s16</th>
<th>p64s8</th>
<th>n32</th>
<th>p32u16</th>
<th>p32u8</th>
<th>s32</th>
<th>p32s16</th>
<th>p32s8</th>
<th>u64_32</th>
<th>s64_32</th>
</tr>
</thead>
</table>

Table 2. Performance of code translated to the MAP1000A running at 200 MHz (ms)

<table>
<thead>
<tr>
<th>Function</th>
<th>Generic C</th>
<th>Native C Intrinsics (not translated)</th>
<th>TriMedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>42.4</td>
<td>1.91</td>
<td>10.4</td>
</tr>
<tr>
<td>invert</td>
<td>14.5</td>
<td>1.28</td>
<td>3.52</td>
</tr>
<tr>
<td>binary dilate</td>
<td>9.47</td>
<td>2.00</td>
<td>6.82</td>
</tr>
<tr>
<td>horizontal flip</td>
<td>13.0</td>
<td>1.28</td>
<td>10.5</td>
</tr>
<tr>
<td>transpose</td>
<td>29.7</td>
<td>1.39</td>
<td>8.37</td>
</tr>
<tr>
<td>histogram generation</td>
<td>25.7</td>
<td>2.48</td>
<td>4.73</td>
</tr>
<tr>
<td>affine warping</td>
<td>75.8</td>
<td>15.5</td>
<td>59.2</td>
</tr>
<tr>
<td>8-tap FIR</td>
<td>126</td>
<td>7.90</td>
<td>18.1</td>
</tr>
<tr>
<td>7x7 convolution</td>
<td>589</td>
<td>25.1</td>
<td>179</td>
</tr>
<tr>
<td>D4 wavelet transform</td>
<td>173</td>
<td>5.69</td>
<td>67.9</td>
</tr>
</tbody>
</table>
Table 3. Performance of code translated to the TriMedia TM1100 running at 133 MHz (ms)

<table>
<thead>
<tr>
<th>Function</th>
<th>Generic C</th>
<th>Native C Intrinsics (not translated)</th>
<th>MAP1000A</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>28.9</td>
<td>2.63</td>
<td>2.91</td>
</tr>
<tr>
<td>invert</td>
<td>14.2</td>
<td>1.14</td>
<td>2.65</td>
</tr>
<tr>
<td>binary dilate</td>
<td>12.6</td>
<td>3.27</td>
<td>5.42</td>
</tr>
<tr>
<td>horizontal flip</td>
<td>11.6</td>
<td>2.24</td>
<td>3.96</td>
</tr>
<tr>
<td>transpose</td>
<td>102</td>
<td>9.32</td>
<td>9.32</td>
</tr>
<tr>
<td>histogram generation</td>
<td>21.5</td>
<td>4.85</td>
<td>12.7</td>
</tr>
<tr>
<td>affine warping</td>
<td>70.6</td>
<td>43.7</td>
<td>130</td>
</tr>
<tr>
<td>8-tap FIR</td>
<td>108</td>
<td>13.1</td>
<td>24.2</td>
</tr>
<tr>
<td>7x7 convolution</td>
<td>860</td>
<td>82.5</td>
<td>98.6</td>
</tr>
<tr>
<td>D4 wavelet transform</td>
<td>272</td>
<td>44.6</td>
<td>44.6</td>
</tr>
</tbody>
</table>

Figure 1. Structure of the intrinsic translation program.
Emulation of hmpv_rs_pu8_ra(value, rs_amt)
/* Shift the partitions in value right by rs_amt */
-------------------------------------------------------
temp1 = 0x0101010101010101 << (rs_amt - 1)
temp2 = hmpv_band_64(value, temp1) /* band is bitwise AND */
temp3 = temp2 >> (rs_amt - 1)
temp4 = hmpv_rs_pu8_td(value, rs_amt) /* rs is right shift */
output = hmpv_add_pu8(temp4, temp3)

Emulation of hmpv_rs_pu8_re(value, rs_amt)
-------------------------------------------------------
temp1 = 0x0101010101010101 << (rs_amt - 1)
temp2 = hmpv_band_64(value, temp1)
temp3 = temp2 >> (rs_amt - 1)
temp4 = hmpv_rs_pu8_td(value, rs_amt)
temp5 = value << (width - (rs_amt - 1))
temp6 = hmpv_min_pu8(0x0101010101010101, temp5) /* mm_min gives the minimum for each partition */
temp7 = hmpv_band_64(temp6, temp3)
output = hmpv_add_pu8(temp4, temp7)

Figure 2. Emulating the MAP fixed-point rounding modes.

Tight loop for 8-bit image addition using universal C intrinsics
-------------------------------------------------------------
dst64_ptr[j] = mm_sa(mm_add(src64_ptr1[j], src64_ptr2[j]));
dst64_ptr[j + 1] = mm_sa(mm_add(src64_ptr1[j + 1], src64_ptr2[j + 1]));

Tight loop for 8-bit image addition using MAP C intrinsics
-------------------------------------------------------------
dst64_ptr[j] = hmpv_add_pu8_sa(src64_ptr1[j], src64_ptr2[j]);
dst64_ptr[j + 1] = hmpv_add_pu8_sa(src64_ptr1[j + 1], src64_ptr2[j + 1]); /* hmpv_add_pu8_sa adds two 64-bit values with 8-bit partition and saturates the result. */

Figure 3. Code fragments demonstrating the use of universal C intrinsics and MAP C intrinsics.
Figure 4. Section of the translation table for translating the universal C intrinsics into the MAP C intrinsics.
Figure 5. Combining the MPI with a universal data flow code generator (UDFCG).